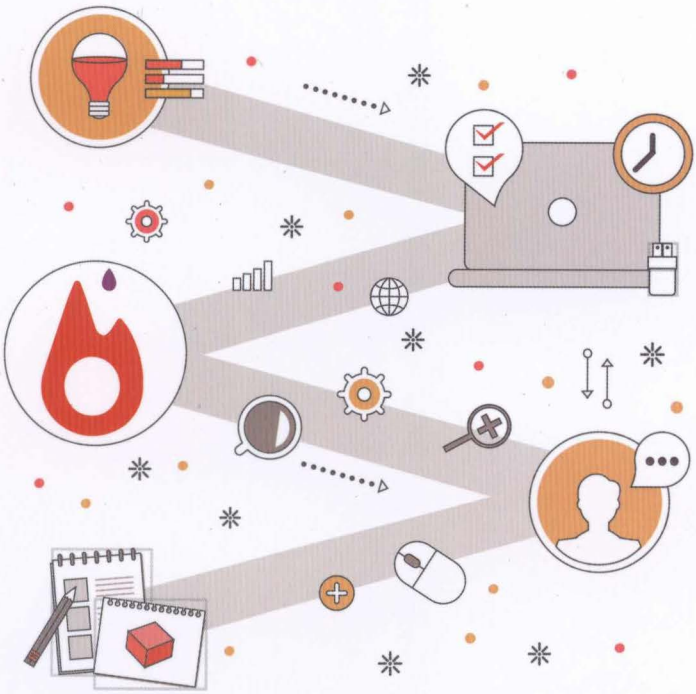
Broadview®
www.broadview.com.cn

博文视点AI系列

本书将带您从零开始进入深度学习这个充满魔力的世界！
本书不仅是深度学习的入门指南，同时也是PyTorch的入门教程。

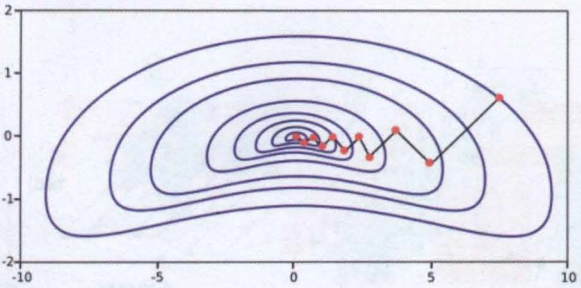


深度学习入门之PyTorch

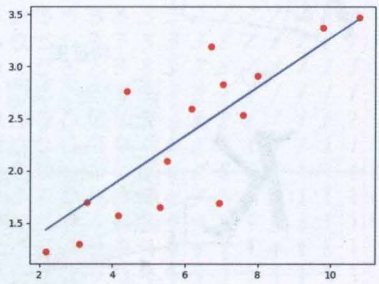
廖星宇 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

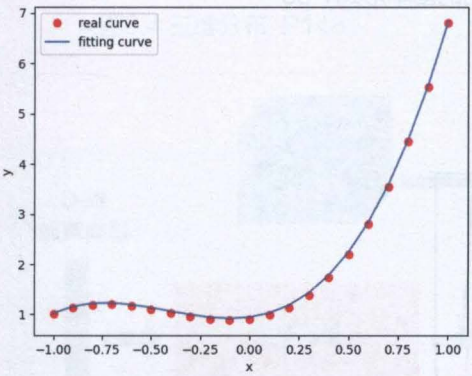
第3章



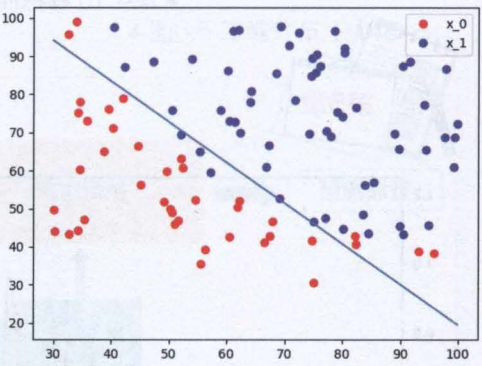
▲图3.1 一阶优化算法 P31



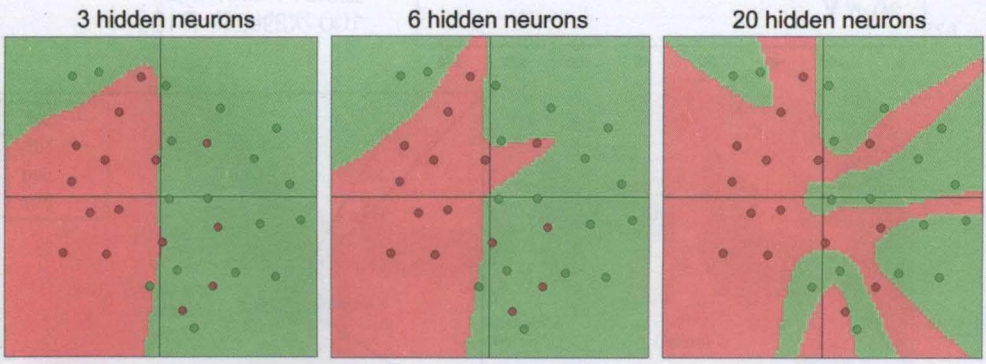
▲图3.5 一元回归 P38



▲图3.7 多项式回归 P41



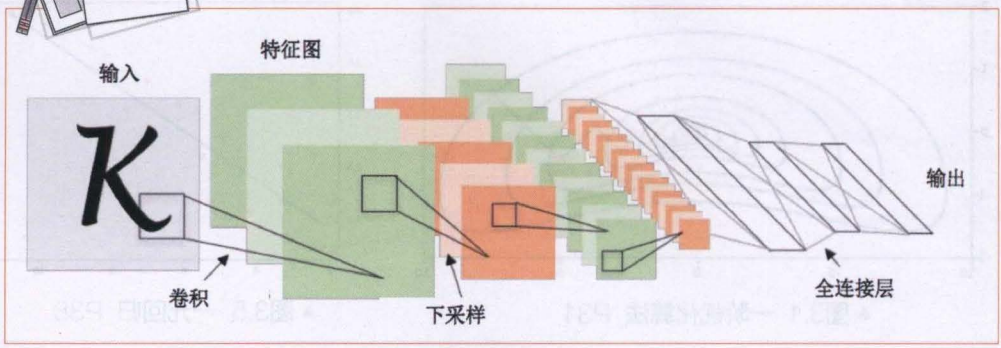
▲图3.12 直线分开两类数据点 P49



▲图3.19 不同网络容量训练结果 P56

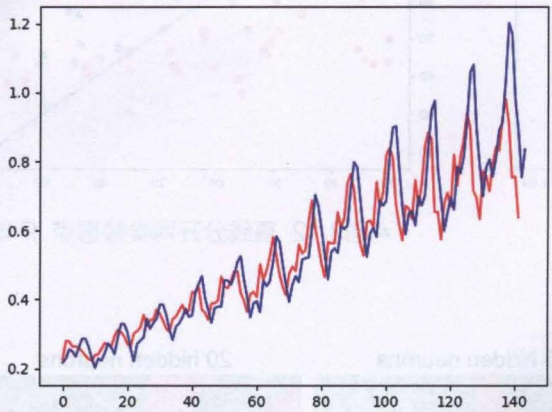
本书实例图

第4章



▲ 图4.10 卷积神经网络的基本形式 P86

第5章



◀ 图5.27 Adam 训练100 次的结果 P130

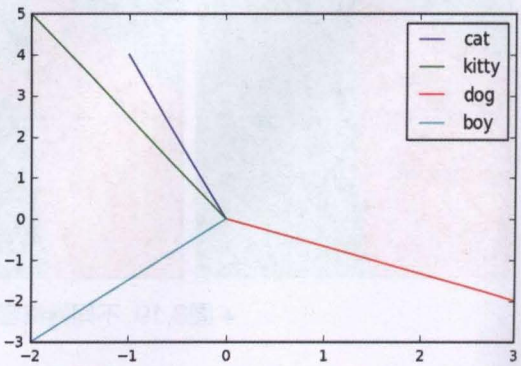
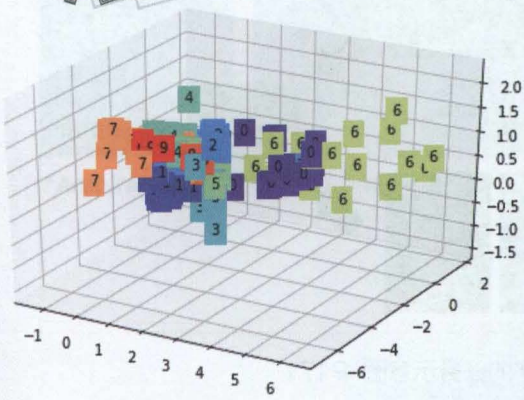
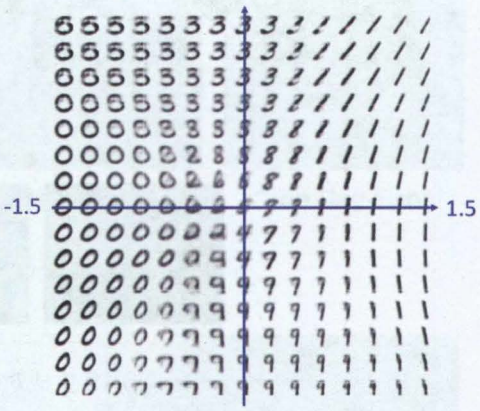


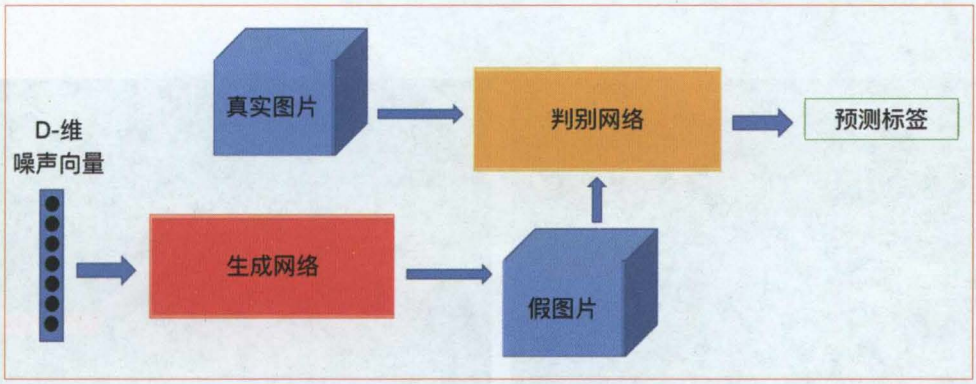
图5.28 不同词向量的夹角 P132 ▶



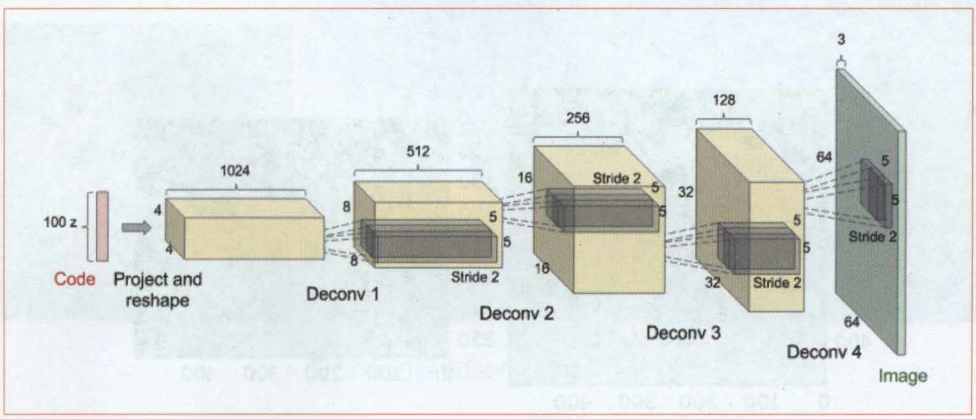
▲图6.4 三维分布 P148



▲图6.5 二维分布 P148

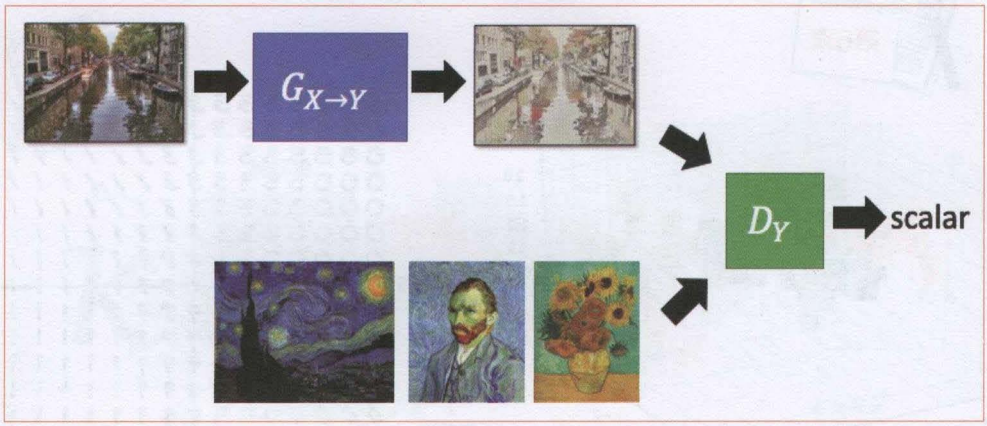


▲图6.8 生成对抗网络生成数据过程 P154



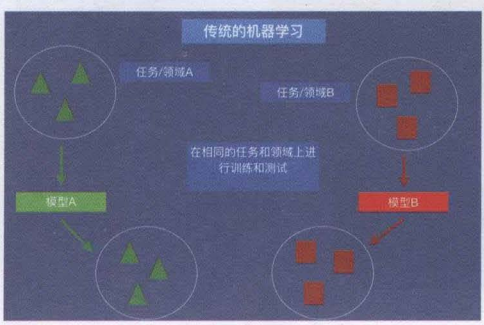
▲图6.9 生成模型 P154

本书实例图

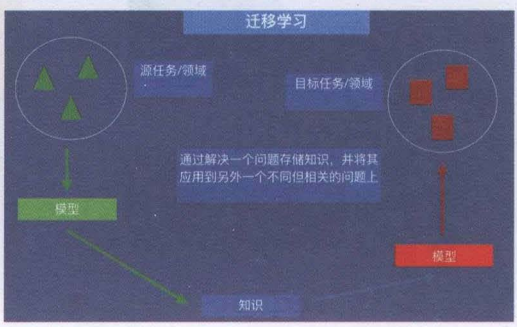


▲图6.19 生成对抗网络的任务示意图 P171

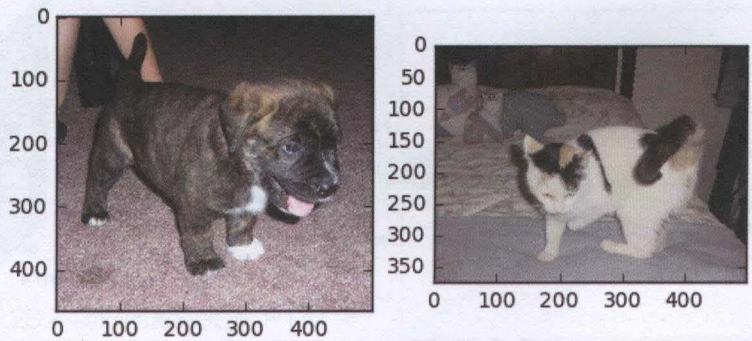
第7章



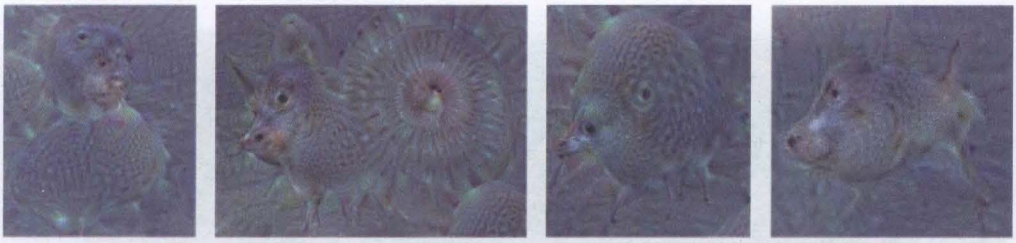
▲图7.2 传统的机器学习 P175



▲图7.3 迁移学习 P176



▲图7.4 迁移学习的模型训练 P178



▲图7.8 云朵梦境 P183

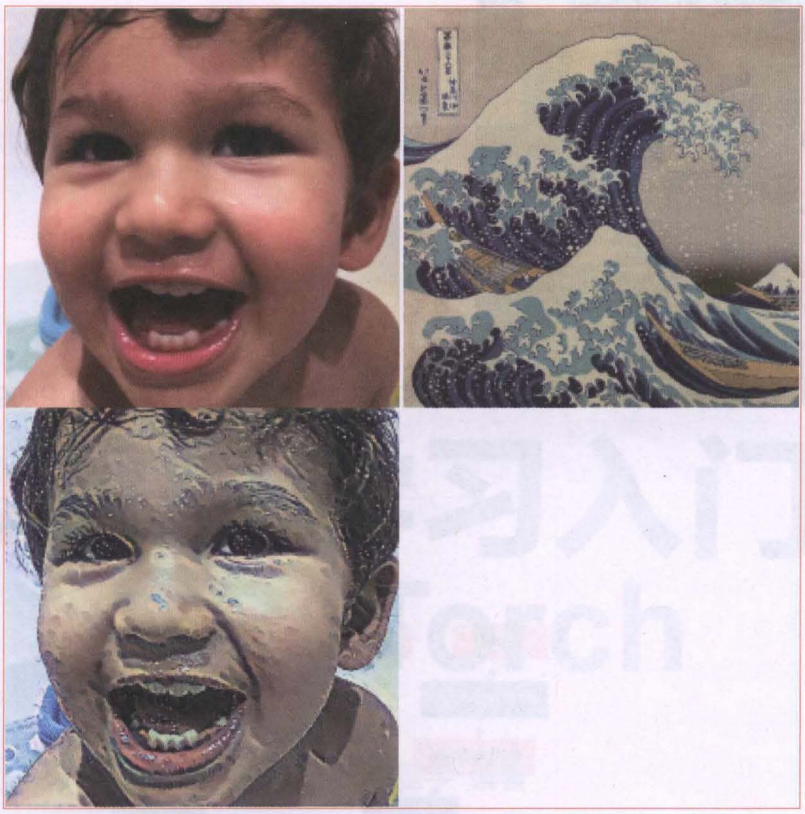
▲图7.15 对云做Deep Dream P194

▲图7.16 对猫做Deep Dream P195

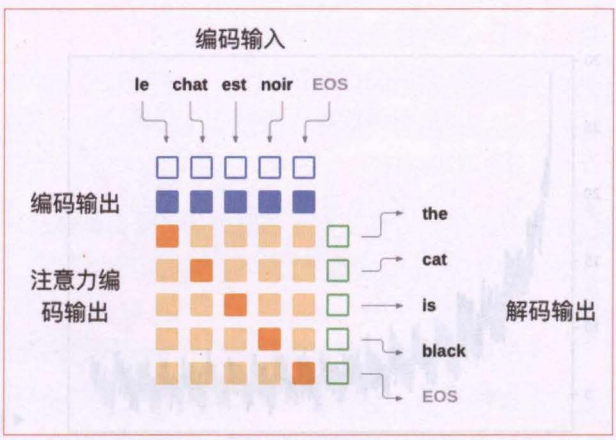
本书实例图

▲图7.17 prisma 效果图 P196

▲图7.18 融合艺术效果 P197

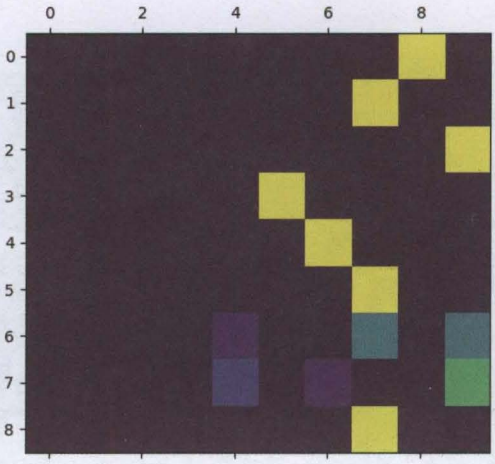


▲图7.19 图片风格融合结果 P205



▲图7.22 注意力机制 P208

本书实例图



◀ 图7.23 与权重结合的注意力机制 P209

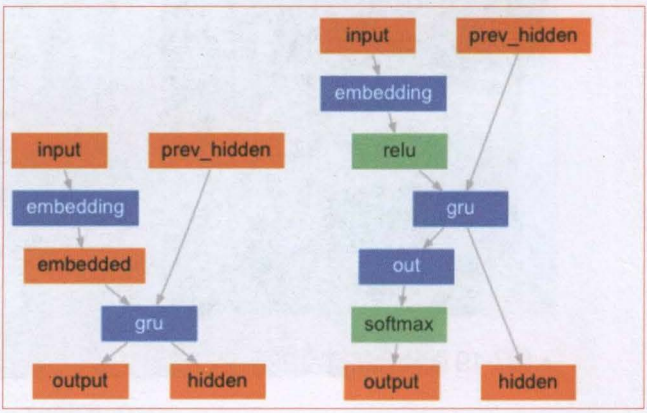
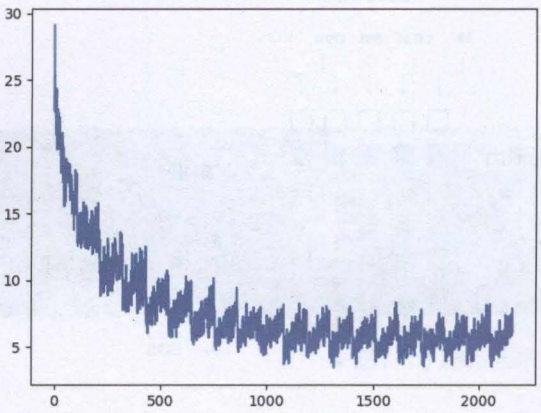


图7.25 编码和解码 P215 ▶



◀ 图7.26 误差效果图 P219

深度学习入门之PyTorch

廖星宇 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

深度学习如今已经成为科技领域最炙手可热的技术，在本书中，我们将帮助你入门深度学习。本书将从机器学习和深度学习的基础理论入手，从零开始学习 PyTorch，了解 PyTorch 基础，以及如何用 PyTorch 框架搭建模型。通过阅读本书，你将学到机器学习中的线性回归和 Logistic 回归、深度学习的优化方法、多层全连接神经网络、卷积神经网络、循环神经网络，以及生成对抗网络，最后通过实战了解深度学习前沿的研究成果，以及 PyTorch 在实际项目中的应用。

本书将理论和代码相结合，帮助读者更好地入门深度学习，适合任何对深度学习感兴趣的人阅读。

图书在版编目（CIP）数据

深度学习入门之 PyTorch / 廖星宇编著. – 北京：电子工业出版社，2017.10
（博文视点 AI 系列）
ISBN 978-7-121-32620-2
I. ①深… II. ①廖… III. ①机器学习 IV. ①TP181
中国版本图书馆 CIP 数据核字（2017）第 215492 号

责任编辑：孙学瑛 sxy@phei.com.cn
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　　印张：14.5　字数：299 千字　　彩插：4
版　　次：2017 年 10 月第 1 版
印　　次：2018 年 3 月第 4 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

前言

随着 AlphaGo 以 3:1 的成绩战胜李世石，人们对人工智能的热情如井喷式增长，你也许对人工智能充满兴趣，向往着了解机器学习，特别是深度学习，那么本书恰好能够由浅及深地带你进入深度学习这个世界。

讲深度学习的书有很多，深度学习的框架也有很多，本书将以 PyTorch 为工具从基础的线性回归开始，讲到时下最前沿的生成对抗网络，并在其中穿插 PyTorch 的教学，所以本书不仅仅是深度学习的入门指南，同时也是 PyTorch 的入门教程。

本书针对的对象是对深度学习有所了解、用过一些深度学习框架（如使用 TensorFlow 跑过简单的模型），但是希望能够用 PyTorch 进行深度学习研究和学习的入门者。阅读本书并不需要太多的数学基础，但是需要一定的 Python 基础。本书中的数学推导不多，感觉困难的读者可以跳过，这对理解全书的主要内容不会造成影响。

本书的主要内容包括：

第 1 章，深度学习介绍；

第 2 章，深度学习框架；

第 3 章，多层全连接神经网络；

第 4 章，卷积神经网络；

第 5 章，循环神经网络；

第 6 章，生成对抗网络；

第 7 章，深度学习实战。

建议读者按照本书的内容顺序学习，因为后面的内容会以前面的内容为基础，另外本书的全部代码放在了 https://github.com/SherlockLiao/code-of-learn-deep-learning-with-pytorch 中，读者可以前往下载。

本书面向的对象是初学者，学习完本书之后，读者能够大致了解深度学习的基本知识，基本掌握 PyTorch 的使用方法，知道如何根据实际问题搭建对应的深层网络结构，并能够进行调参得到较好的结果。当然本书只是一本入门读物，如果希望以后从事该领域的研究，仅靠此书是不够的，需要阅读更多专业的书籍和学术论文。

在本书的创作过程离不开很多人对我的帮助，书中的一部分内容参考了李飞飞教授在斯坦福大学开设的课程 cs231n，以及台湾国立大学教授李宏毅开始的 MLDS，除此之外还参考了网络上的一些图例，因为大多找不到出处，所以无法一一列出进行感谢。

除此之外，还感谢在写书的过程中我的家人对我的鼓励和信任，正是他们的支持让我能够坚持写完整本书。

最后，感谢电子工业出版社给我这次机会让我能够出版此书，同时也感谢孙学瑛编辑全程对我的帮助。

由于本人水平有限，书中存在的纰漏，欢迎大家向我指出，我也很高兴收到大家的意见和建议，不胜感激。

廖星宇

中国科学技术大学数学系

E-mail：sherlockliao01@gmail.com

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误：**您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32620

目录

第 1 章

深度学习介绍

自古以来，发明家们都梦想着能够创造有思想的机器人，当电脑问世之后，人们一直想知道它们是否可以变得更加智能。如今人工智能成为一个有着无限商业价值和研究价值的领域，人们也看到了人工智能在图像、语音、自然语言上取得的巨大成功。

这一章，我们将简要地介绍一下人工智能及其重要性，以及什么是数据挖掘、机器学习和深度学习，并介绍它们彼此之间的联系与区别，其中会重点介绍深度学习，最后给出一些合理可行的学习建议，帮助大家从零开始进入深度学习这个充满魔力的领域。

1.1 人工智能

人工智能（Artificial Intelligence），也称为机器智能，是指由人工制造出来的系统所表现的智能，所谓的智能，即指可以观察周围环境并据此做出行动以达到目的。

在人工智能的早期，那些对人类智力来说非常困难、但对计算机来说相对简单的问题迅速得到解决，比如，那些可以通过一系列形式化的数学规则来描述的问题。AI 的真正挑战在于解决那些对人来说很容易执行、但很难形式化描述的任务，比如，识别人们所说的话或图像中的脸。对于这些问题，我们人类往往可以凭借直觉轻易地解决，因为我们已经在上万年的进化中形成了这些直觉性的能力，但是机器却很难找到实现的方法。

长久以来人们一直相信人工智能是存在的，但是却不知道如何实现。以前的科幻电影总会融入人工智能，比如《星球大战》《终结者》《骇客帝国》，等等。电影的渲染使我们总觉得人工智能缺乏真实感，或者总将人工智能和机器人联系在一起。其实我

们身边早已实现了一些弱人工智能，只是因为人工智能听起来很神秘，所以我们往往没有意识到。

首先，不要一提到人工智能就想到机器人。机器人只是人工智能的一种容器，如果将人工智能比作大脑，那么机器人就好似身体——然而这个身体却不是必需的，比如现在很火的 AlphaGo，其背后充满着软件、算法和数据，它下围棋是一种人格化的体现，然而其本身并没有“机器人”这个硬件形式。

人工智能的概念很宽泛，现在根据人工智能的实力将它分成三大类。

（1）弱人工智能（Artificial Narrow Intelligence，ANI）

弱人工智能是擅长于单个方面的人工智能。比如战胜世界围棋冠军的人工智能 AlphaGo，它只会下围棋，如果你让他辨识一下猫和狗，它就不知道怎么做了。我们现在实现的几乎全是弱人工智能。

（2）强人工智能（Artificial General Intelligence，AGI）

这是类似人类级别的人工智能。强人工智能是指在各方面都能和人类比肩的人工智能，人类能干的脑力活，它都能干。创造强人工智能比创造弱人工智能难得多，我们现在还做不到。Linda Gottfredson 教授把智能定义为“一种宽泛的心理能力，能够进行思考、计划、解决问题、抽象思维、理解复杂理念、快速学习和从经验中学习等操作。”强人工智能在进行这些操作时应该和人类一样得心应手。

（3）超人工智能（Artificial Superintelligence，ASI）

牛津哲学家、知名人工智能思想家 Nick Bostrom 把超级智能定义为“在几乎所有领域都比最聪明的人类大脑都聪明很多，包括科学创新、通识和社交技能。”超人工智能可以是各方面都比人类强一点，也可以是各方面都比人类强万亿倍的。

我们现在处于一个充满弱人工智能的世界，比如垃圾邮件分类系统，是一个可以帮助我们筛选垃圾邮件的弱人工智能；Google 翻译是一个可以帮助我们翻译英文的弱人工智能；AlphaGo 是一个可以战胜世界围棋冠军的弱人工智能，等等。这些弱人工智能算法不断地加强创新，每一个弱人工智能的创新，都是在给通往强人工智能和超人工智能的旅途添砖加瓦。正如人工智能科学家 Aaron Saenz 所说，现在的弱人工智能就像地球早期软泥中的氨基酸，可能突然之间就形成了生命。

1.2 数据挖掘、机器学习与深度学习

大数据的兴起使得数据科学家作为一种新生职业被提出，数据研究高级科学家 Rachel Schutt 将其定义为“计算机科学家、软件工程师和统计学家的混合体”。数据

挖掘作为一个学术领域，横跨多个学科，涵盖但不限于统计学、数学、机器学习和数据库，此外还运用在各类专业领域，比如油田、电力、海洋生物、历史文本、图像、电子通信等。

1.2.1 数据挖掘

简单来说，数据挖掘就是在大型的数据库中发现有用的信息，并加以分析的过程，也就是人们所说的 KDD（knowledge discovery in database）。一个数据的处理过程，就是从输入数据开始，对数据进行预处理，包括特征选择、规范化、降低维数、数据提升等，然后进行数据的分析和挖掘，再经过处理，例如模式识别、可视化等，最后形成可用信息的全过程。

所以说数据挖掘只是一种概念，即从数据中挖掘到有意义的信息，从大量的数据中寻找数据之间的特性。

1.2.2 机器学习

机器学习算是实现人工智能的一种途径，它和数据挖掘有一定的相似性，也是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、计算复杂性理论等多门学科。对比于数据挖掘从大数据之间找相互特性而言，机器学习更加注重算法的设计，让计算机能够自动地从数据中“学习”规律，并利用规律对未知数据进行预测。因为学习算法涉及了大量的统计学理论，与统计推断联系尤为紧密，所以也被称为统计学习方法。

机器学习可以分为以下五个大类：

（1）监督学习：从给定的训练数据集中学习出一个函数，当新的数据到来时，可以根据这个函数预测结果。监督学习的训练集要求是输入和输出，也可以说是特征和目标。训练集中的目标是由人标注的。常见的监督学习算法包括回归与分类。

（2）无监督学习：无监督学习与监督学习相比，训练集没有人为标注的结果。常见的无监督学习算法有聚类等。

（3）半监督学习：这是一种介于监督学习与无监督学习之间的方法。

（4）迁移学习：将已经训练好的模型参数迁移到新的模型来帮助新模型训练数据集。

（5）增强学习：通过观察周围环境来学习。每个动作都会对环境有所影响，学习对象根据观察到的周围环境的反馈来做出判断。

传统的机器学习算法有以下几种：线性回归模型、logistic 回归模型、k-临近算法、决策树、随机森林、支持向量机、人工神经网络、EM 算法、概率图模型等。

1.2.3 深度学习

深度学习的最初级版本是人工神经网络，这是机器学习的一个分支，其试图模拟人脑，通过更加复杂的结构自动提取数据特征。

在深度学习发展起来之前，机器学习虽然发展了几十年，但还是存在很多人工智能没法良好解决的问题，例如图像识别、语音识别、自然语言处理等，而深度学习的出现则很好地解决了这些领域的一部分问题。正因为大数据的兴起和高性能 GPU 的出现，使得更加复杂的网络模型成为可能，这也促使深度学习有了进一步的发展。

1. 深度学习的历史

首先通过一张图来概括一下深度学习的历史浪潮，如图 1.1所示。

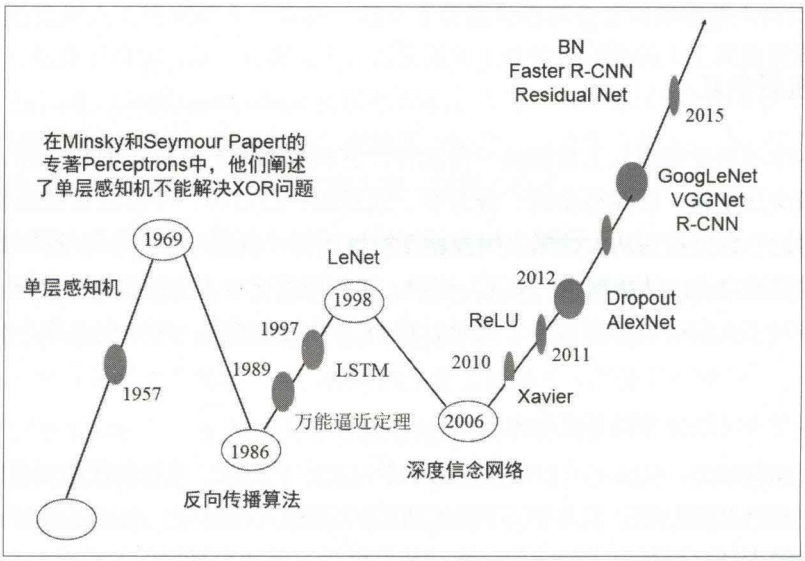


图 1.1　深度学习的发展史

通过这张图，我们可以很明显地看出深度学习从发展到崛起经历了两个低谷，这两个低谷也将深度学习的发展分为了三个不同的阶段，下面分别讲述这三段历史。

第一代神经网络（1958 年—1969 年）

最早的神经网络的思想起源于 1943 年的 MCP 人工神经元模型，当时人们希望能够用计算机来模拟人的神经元反应的过程，该模型将神经元简化为三个过程：输入信号线性加权，求和，非线性激活（阈值法），如图1.2所示。

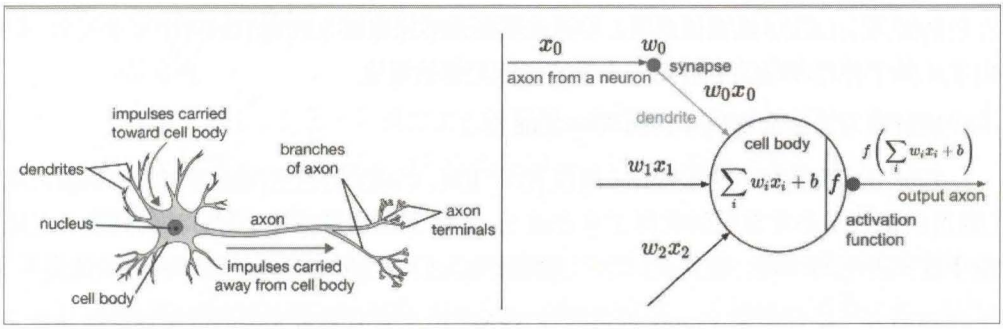


图 1.2 神经元模型

第一次将 MCP 用于机器学习（分类）的当属 1958 年 Rosenblatt 发明的感知器（perceptron）算法。该算法使用 MCP 模型对输入的多维数据进行二分类，且能够使用梯度下降法从训练样本中自动学习更新权值。1962 年，该方法被证明为能够收敛，它的理论与实践效果引发了第一次神经网络的浪潮。

然而学科发展的历史不总是一帆风顺的。

1969 年，美国数学家及人工智能先驱 Minsky 在其著作中证明了感知器本质上是一种线性模型，只能处理线性分类问题，就连最简单的 XOR（亦或）问题都无法正确分类。这等于直接宣判了感知器的死刑，神经网络的研究也陷入了近 20 年的停滞。

第二代神经网络（1986 年—1998 年）

第一次打破非线性诅咒的当属现代深度学习大牛 Hinton，他在 1986 年发明了适用于多层感知器（MLP）的 BP 算法，并采用 Sigmoid 进行非线性映射，有效解决了非线性分类和学习的问题。该方法引发了神经网络的第二次热潮。

1989 年，Robert Hecht-Nielsen 证明了 MLP 的万能逼近定理，即对于任何闭区间内的一个连续函数 f，都可以用含有一个隐含层的 BP 网络来逼近。该定理的发现，极大地鼓舞了神经网络的研究人员。

也是在 1989 年，LeCun 发明了卷积神经网络——LeNet，并将其用于数字识别，且取得了较好的成绩，不过当时并没有引起足够的注意。

值得强调的是，1989 年以后由于没有提出特别突出的方法，且神经网络一直缺少相应的严格的数学理论支持，神经网络的热潮渐渐冷淡下去。冰点发生于 1991 年，BP 算法被指出存在梯度消失问题，即在误差梯度后向传递的过程中，后层梯度以乘性方式叠加到前层，由于 Sigmoid 函数的饱和特性，后层梯度本来就小，误差梯度传到前层时几乎为 0，因此无法对前层进行有效的学习，该发现对此时的神经网络发展无异于雪上加霜。

1997 年，LSTM 模型被发明，尽管该模型在序列建模上表现出的特性非常突出，但由于正处于神经网络的下坡期，也没有引起足够的重视。

统计学习方法的春天（1986 年—2006 年）

1986 年，决策树方法被提出，很快 ID3，ID4，CART 等改进的决策树方法相继出现（到目前仍然是非常常用的机器学习方法），这些方法也是符号学习方法的代表。正是由于这些方法的出现，使得统计学习开始进入人们的视野，迎来统计学习方法的春天。

1995 年，统计学家 Vapnik 提出线性 SVM。由于它有非常完美的数学理论推导做支撑（统计学与凸优化等），并且非常符合人的直观感受（最大间隔），逐渐成为当时的主流算法，更重要的是它在线性分类的问题上取得了当时最好的成绩，这使得神经网络更陷入无人问津的境地。

1997 年，AdaBoost 被提出，该方法是 PAC（Probably Approximately Correct）理论在机器学习实践上的代表，也催生了集成学习（Ensemble Learning）这一类方法的诞生，在回归和分类任务上取得了非常好的效果。该方法通过一系列的弱分类器集成，达到强分类器的效果。现在集成学习仍然活跃在传统机器学习方法中，在很多比赛中依旧大放异彩。

2000 年，Kernel SVM 被提出，核化的 SVM 通过一种巧妙的方式将原空间线性不可分的问题，通过 Kernel 映射成高维空间的线性可分问题，成功解决了非线性分类的问题，且分类效果非常好。至此也更加终结了神经网络时代，在有着如此多完美理论支持的方法中，神经网络似乎被宣告了死刑。

2001 年，随机森林被提出，这是集成方法的另一代表，该方法的理论扎实，比 AdaBoost 能更好地抑制过拟合问题，实际效果也非常不错，使得机器学习更向前迈进一步。

2001 年，一种新的统一框架——图模型被提出，该方法试图统一机器学习混乱的方法，如朴素贝叶斯、SVM、隐马尔可夫模型等，为各种学习方法提供一个统一的描述框架，希望实现大一统的理论框架。

第三代神经网络深度学习（2006 年至今）

该阶段又分为两个时期：快速发展期（2006 年—2012 年）与爆发期（2012 年至今）。

- **快速发展期（2006 年—2012 年）**

  2006 年，深度学习元年。这一年，Hinton 提出了深层网络训练中梯度消失问题的解决方案：“无监督预训练对权值进行初始化 + 有监督训练微调”。其主要思想是先通过自学习的方法学习到训练数据的结构（自动编码器），然后在该结

构上进行有监督训练微调。但是由于没有特别有效的实验验证，该论文并没有引起重视。

2011 年，ReLU 激活函数被提出，该激活函数能够有效地抑制梯度消失的问题。

2011 年，微软首次将深度学习应用在语音识别上，取得了重大突破。

- **爆发期（2012 年至今）**

  2012 年，Hinton 课题组为了证明深度学习的潜力，首次参加 ImageNet 图像识别比赛，通过构建 CNN 网络 AlexNet 一举夺得冠军，并且碾压了第二名（SVM 方法）的分类性能。也正是由于该比赛，CNN 吸引了众多研究者的注意。

AlexNet 的创新点主要有以下几点：

（1）首次采用 ReLU 激活函数，极大加快了收敛速度，并且从根本上解决了梯度消失的问题；

（2）由于 ReLU 方法可以很好地抑制梯度消失问题，AlexNet 抛弃了“预训练 + 微调”的方法，完全采用有监督训练。也正因为如此，现在深度学习的主流学习方法也变成了纯粹的有监督学习；

（3）扩展了 LeNet5 结构，添加 Dropout 层减小过拟合，LRN 层增强泛化能力/减小过拟合；

（4）首次采用 GPU 对计算进行加速。

2013 年、2014 年、2015 年，通过 ImageNet 图像识别比赛，深度学习因其网络结构、训练方法、GPU 硬件不断进步，在其他领域也不断地征服战场。

2015 年，Hinton，LeCun，Bengio 论证了局部极值问题对于深度学习的影响，得到的结果是 Loss 的局部极值问题对于深层网络的影响可以忽略，该论断也消除了笼罩在神经网络上的局部极值问题的阴霾。具体原因是深层网络虽然局部极值非常多，但是通过深度学习的 Batch Gradient Descent 优化方法很难陷进去，而且就算陷进去，其局部极小值点与全局极小值点也是非常接近的。而浅层网络却虽然拥有较少的局部极小值点，但是却很容易陷进去，且这些局部极小值点与全局极小值点相差较大。对于这一点，论述原文其实没有严格地证明，只是简单叙述。

2015 年，何恺明提出 Deep Residual Net。分层预训练时，ReLU 和 BatchNormalization 都是为了解决深度神经网络优化时的梯度消失或者爆炸问题，但是在对更深层的神经网络进行优化时，又出现了新的 Degradation 问题，“即通常来说，如果在 VGG16 后面加上若干个单位映射，网络的输出特性将和 VGG16 一样，这说明更深层的网络的潜在分类性能只可能大于等于 VGG16 的性能，不可能变坏，然而实际效果却是：如果简单地加深 VGG16 的话，分类性能就会下降（不考虑模型过拟合问题）”。

Residual 网络认为这说明深度学习网络在学习单位映射方面有困难，因此设计了一个对于单位映射（或接近单位映射）有较强学习能力的深度学习网络，极大地增强了深度学习网络的表达能力。使用此方法能够轻松地训练高达 150 层的网络，使得深度学习成为了名副其实的“深度”学习。

2. 深度学习结构

随着神经网络的发展，目前比较流行的网络结构分别有：深度神经网络（DNN）、卷积神经网络（CNN）、循环递归神经网络（RNN）、生成对抗网络（GAN），等等，本书会逐一详细介绍，在此不再赘述。

1.3 学习资源与建议

随着近年深度学习的兴起，很多研究者都投入这个领域当中，由于各个大学都将自己的课程放到了网上，出现了很多学习资源和网络课程，而且很多大公司如 Google，Facebook 都将自己的开源框架放到了 Github 上，使得深度学习的入门越来越简单，很多麻烦的重复操作已经通过框架简化了，这也让我们每个人都能有机会接触深度学习。

网上有各种各样的学习经验分享，有的人注重理论知识的积累，看了很多书，但是动手实践的经验为 0；也有一些人热衷于代码的实现，每天学习别人已经写好的代码。对于这两种情况，我认为都是不好的，深度学习是理论和工程相结合的领域，不仅仅需要写代码的能力强，也需要有理论知识能够看懂论文，实现论文提出来的新想法，所以我们的学习路线应该是理论与代码相结合，平衡两边的学习任务，不能出现只管一边而不学另外一边的情况，因为只有理论与代码兼顾才不至于一旦学习深入，就会发现自己会有很多知识的漏洞。

在学习本书之前，需要一定的 Python 语言基础和微积分、线性代数的基础，下面将给出开始阅读本书之前的学习建议，以及读完本书后继续深入了解深度学习领域的学习建议。

在开始本书之前，对于微积分和线性代数需要掌握的知识并不多，对于微积分只需要知道导数和偏导数，对于线性导数只需要知道矩阵乘法就可以了。

对于 Python 语言，给出三个学习资源，学完第一个学习资源之后就可以没有障碍地阅读完本书，后续的两个学习资源帮助你更深入地了解 Python 语言及其数值计算。

（1）《笨方法学 Python》(*Learn Python the Hard Way*)

本书面向零基础的学者，通过一系列简单的例子快速入门 Python 的基本操作。

（2）廖雪峰的 Python 入门

这个系列教程可用来更全面地学习 Python，掌握前几章的 Python 基础即可，后面讲授的部分是更为专业的 Web 开发的内容，对于机器学习而言不需要掌握这些部分。

（3）*Edx: Introduction to Computer Science and Programming Using Python*

这是 MIT 的公开课，以 Python 作为入门语言，简洁、全面地讲述了计算机科学的内容，适合更进一步的学习。

以上的 Python 入门课程只需要看完第一个学习资源便可开始本书的阅读。

下面给出读完本书，以及在本书的阅读过程中的一些参考学习资源。

（1）线性代数

线性代数相当于深度学习的基石，深度学习里面有着大量的矩阵运算，而且线性代数的一些矩阵分解的思想也被借鉴到了机器学习中，所以必须熟练掌握线性代数。可参考以下资源学习：

- 《线性代数应该这样学》(*Linear Algebra done right*)
- MIT 的线性代数公开课
- *Coding The Matrix*

（2）机器学习基础

虽然深度学习现在很火，但是也需要掌握其根本，即机器学习，这才是本质与核心。

这里的学习资源从易到难排列：

- Coursera 上 Andrew Ng 的机器学习入门课程
- 林轩田的机器学习基石和机器学习技法
- Udacity 的机器学习纳米学位
- 周志华著的《机器学习》
- 李航著的《统计学习方法》
- *Pattern Recognition and Machine Learning*

（3）深度学习

这是最近几年最为活跃的研究领域，爆发了很多革命性的突破，很多前沿的学习资源，如：

- Udacity 的两个深度学习课程
- Coursera 的 *Neural Networks for Machine Learning*
- Stanford 的 cs231n

- Stanford 的 cs224n

以上的学习资源很多，可以学习及参考，很多课程都是理论与实践相结合的，通过上面资源的学习，可以完成深度学习领域绝大部分的知识储备。

第 2 章

深度学习框架

在开始深度学习项目前，选择一个合适的框架是非常重要的事情，因为选择一个合适的框架能让你事半功倍。目前研究者使用了各种不同的框架来达到他们的研究目的，也从侧面印证了如今深度学习领域百花齐放。

这一章，我们将介绍一下各个大公司的深度学习开源框架，随后着重介绍本书的主角 PyTorch，最后介绍一下如何安装 PyTorch，将你的电脑配置成一台能够进行深度学习的机器。

2.1 深度学习框架介绍

在深度学习初始阶段，每个深度学习研究者都需要写大量的重复代码。为了提高工作效率，这些研究者就将这些代码写成了一个框架放到网上让所有研究者一起使用，接着网上就出现了不同的框架，随着时间的推移，最为好用的几个框架被大量的人使用从而流行了起来，首先让我们来介绍一下目前全世界最为流行的几大深度学习框架。

Tensorflow

首先要介绍的当然是 Google 开源的 TensorFlow，这是一款使用 C++ 语言开发的开源数学计算软件，使用数据流图（Data Flow Graph）的形式进行计算。图中的节点代表数学运算，而图中的线条表示多维数据数组（tensor）之间的交互。TensorFlow 灵活的架构可以部署在一个或多个 CPU、GPU 的台式及服务器中，或者使用单一的 API 应用

在移动设备中。TensorFlow 最初是由研究人员和 Google Brain 团队针对机器学习和深度神经网络进行研究而开发的，目前开源之后几乎可以在各种领域适用。

目前 Tensorflow 是全世界使用人数最多、社区最为庞大的一个框架，因为 Google 公司出品，所以维护与更新也比较频繁，并且有着 Python 和 C++ 的接口，教程也非常完善，同时很多论文复现的第一个版本都是基于 Tensorflow 写的，所以是深度学习界框架默认的老大。

由于其语言太过于底层，目前有很多基于 Tensorflow 的第三方抽象库将 Tensorflow 的函数进行封装，使其变得简洁，目前比较有名的几个是 Keras，Tflearn，tfslim，以及 TensorLayer。

Caffe

和 Tensorflow 名气一样大的是深度学习框架 Caffe，由加州大学伯克利的 Phd 贾扬清开发，全称是 Convolutional Architecture for Fast Feature Embedding，是一个清晰而高效的开源深度学习框架，目前由伯克利视觉学中心（Berkeley Vision and Learning Center，BVLC）进行维护。

从它的名字就可以看出其对于卷积网络的支持特别好，同时也是用 C++ 写的，但是并没有提供 Python 接口，只提供了 C++ 的接口。

Caffe 之所以流行，是因为之前很多 ImageNet 比赛里面使用的网络都是用 Caffe 写的，所以如果你想使用这些比赛里面的网络模型就只能使用 Caffe，这也就导致了很多人直接转到 Caffe 这个框架下面。

Caffe 的缺点是不够灵活，同时内存占用高，只提供了 C++ 的接口，目前 Caffe 的升级版本 Caffe2 已经开源了，修复了一些问题，同时工程水平得到了进一步提高。

Theano

Theano 于 2008 年诞生于蒙特利尔理工学院，其派生出了大量深度学习 Python 软件包，最著名的包括 Blocks 和 Keras。Theano 的核心是一个数学表达式的编译器，它知道如何获取你的结构，并使之成为一个使用 numpy、高效本地库的高效代码，如 BLAS 和本地代码（C++）在 CPU 或 GPU 上尽可能快地运行。它是为深度学习中处理大型神经网络算法所需的计算而专门设计的，是这类库的首创之一（发展始于 2007 年），被认为是深度学习研究和开发的行业标准。

但是目前开发 Theano 的研究人员大多去了 Google 参与 Tensorflow 的开发，所以某种程度来讲 Tensorflow 就像 Theano 的孩子。

Torch

Torch 是一个有大量机器学习算法支持的科学计算框架，其诞生已经有十年之久，但是真正起势得益于 Facebook 开源了大量 Torch 的深度学习模块和扩展。Torch 的特点在于特别灵活，但是另外一个特殊之处是采用了编程语言 Lua，在目前深度学习大部分以 Python 为编程语言的大环境之下，一个以 Lua 为编程语言的框架有着更多的劣势，这一项小众的语言增加了学习使用 Torch 这个框架的成本。

本书的主角 PyTorch 的前身便是 Torch，其底层和 Torch 框架一样，但是使用 Python 重新写了很多内容，不仅更加灵活，支持动态图，也提供了 Python 接口。

MXNet

MXNet 的主要作者是李沐，最早就是几个人抱着纯粹对技术和开发的热情做起来的兴趣项目，如今成为了亚马逊的官方框架，有着非常好的分布式支持，而且性能特别好，占用显存低，同时其开放的语言接口不仅仅有 Python 和 C++，还有 R，Matlab，Scala，JavaScript，等等，可以说能够满足使用任何语言的人。

但是 MXNet 的缺点也很明显，教程不够完善，使用的人不多导致社区不大，同时每年很少有比赛和论文是基于 MXNet 实现的，这就使得 MXNet 的推广力度和知名度不高。

2.2 PyTorch 介绍

2.2.1 什么是 PyTorch

通过前面一节我们已经了解到了几款主流的深度学习框架，下面我们将重点介绍本书的主角——PyTorch。

PyTorch 是 Torch7 团队开发的，从它的名字就可以看出，其与 Torch 的不同之处在于 PyTorch 使用了 Python 作为开发语言。所谓“Python first”，同样说明它是一个以 Python 优先的深度学习框架，不仅能够实现强大的 GPU 加速，同时还支持动态神经网络，这是现在很多主流框架比如 Tensorflow 等都不支持的。

PyTorch 既可以看做加入了 GPU 支持的 numpy，同时也可以看成一个拥有自动求导功能的强大的深度神经网络，除了 Facebook 之外，它还已经被 Twitter、CMU 和 Salesforce 等机构采用，如图2.1所示。

图 2.1　PyTorch 使用厂商

2.2.2　为何要使用 PyTorch

面对如此多的深度学习框架，我们为何要选择 PyTorch 呢？Tensorflow 不是深度学习框架默认的老大吗，为什么不直接选择 Tensorflow 而是要选择 PyTorch 呢？下面分 4 个方面来介绍为何要使用 PyTorch。

（1）掌握一个框架并不能一劳永逸，现在深度学习并没有谁拥有绝对的垄断地位，就算是 Google 也没有，所以只学习 Tensorflow 并不够。同时现在的研究者使用各个框架的都有，如果你要去看他们实现的代码，至少也需要了解他们使用的框架，所以多学一个框架，以备不时之需。

（2）Tensorflow 与 Caffe 都是命令式的编程语言，而且是静态的，首先必须构建一个神经网络，然后一次又一次使用同样的结构，如果想要改变网络的结构，就必须从头开始。但是对于 PyTorch，通过一种反向自动求导的技术，可以让你零延迟地任意改变神经网络的行为，尽管这项技术不是 PyTorch 独有，但目前为止它实现是最快的，能够为你任何疯狂想法的实现获得最高的速度和最佳的灵活性，这也是 PyTorch 对比 Tensorflow 最大的优势。

（3）PyTorch 的设计思路是线性、直观且易于使用的，当你执行一行代码时，它会忠实地执行，并没有异步的世界观，所以当你的代码出现 Bug 的时候，可以通过这些信息轻松快捷地找到出错的代码，不会让你在 Debug 的时候因为错误的指向或者异步和不透明的引擎浪费太多的时间。

（4）PyTorch 的代码相对于 Tensorflow 而言，更加简洁直观，同时对于 Tensorflow 高度工业化的很难看懂的底层代码，PyTorch 的源代码就要友好得多，更容易看懂。深入 API，理解 PyTorch 底层肯定是一件令人高兴的事。一个底层架构能够看懂的框架，你对其的理解会更深。

最后，我们简要总结一下 PyTorch 的特点：

- 支持 GPU；
- 动态神经网络；
- Python 优先；
- 命令式体验；
- 轻松扩展。

拥有着如此多优点的.PyTorch 也有着它的缺点，首先 PyTorch 于 2017 年 3 月开源发布，目前还是 beta 测试版，没有发布正式版本，所以可能有一些小的 Bug；其次因为这款框架比较新，所以使用的人也就比较少，这也就使得它的社区没有那么强大，但是 PyTorch 提供了一个官方的论坛，大多数碰到的问题都可以去里面搜索，里面的答案一般都是由作者或者其他 PyTorch 使用者提供的，论坛的更新也特别频繁，同时也可以去 Github 上面提 Issue，一般很快就会得到开发者的回应，也算是一定程度上解决了社区的问题。

2.3 配置 PyTorch 深度学习环境

通过上面两节，我们简要地了解了现今深度学习的主流框架，以及 PyTorch，下面就开始打造 PyTorch 的深度学习环境。

2.3.1 操作系统的选择

首先需要选择电脑的操作系统，现在一般有三种操作系统：Windows、Mac 和 Linux，这里只能在 Mac 和 Linux 之间选择，这是因为目前 PyTorch 只支持这两种操作系统，Windows 操作系统的支持应该会在未来实现，但是现在还不支持，所以如果要学习 PyTorch，就只能选择 Mac 和 Linux。

如果你有一台苹果电脑，那么你就可以直接开始下面 Python 环境的配置了；如果你是一台 Windows 笔记本或者台式机，这里有两种选择：第一，安装 Linux 虚拟机；第二，安装一个双系统。这里推荐安装双系统，因为虚拟机下的体验不是很好，双系统也

不需要占用太多的电脑空间。对于 Linux 系统，推荐使用 Ubuntu，在网上能够找到很多的安装教程，同时使用者较多，如果遇到问题，容易在网上找到很多解答办法。

2.3.2 Python 开发环境的安装

接下来我们需要一个好用的 Python 的开发环境，对于 Python 而言，比较头疼的就是不同版本环境管理与包管理。针对这些问题，有不少发行版的 Python，比如 WinPython 和 Anaconda 等，这些发行版将 Python 常用的包打包，方便使用者直接使用而不需要再重新安装，这里推荐使用 Anaconda 软件管理包。

那么 Anaconda 是什么呢？Anaconda 是一个用于科学计算的 Python 发行版，支持 Linxu、Mac 和 Windows 系统，提供了包管理与环境管理的功能，可以很方便地解决多版本 Python 并存、切换，以及各种第三方包安装的问题。

Anaconda 的下载可以进入其官网如图2.2所示的 https://www.continuum.io/ downloads。

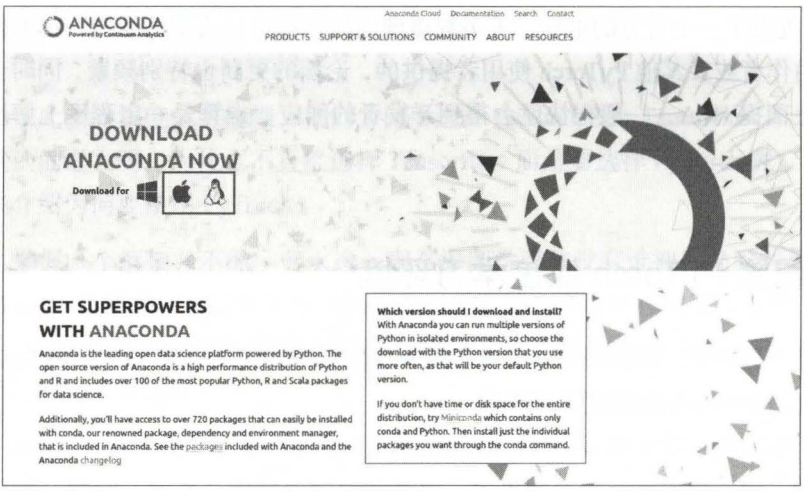


图 2.2 Anaconda 官网

单击图 2.2 上的框来选择 Mac 或者 Linux，进入之后可以看到如图2.3所示的界面。在图 2.3 上的框中选择 Python 版本，注意这里是默认系统为 64 位版本的，一般现在的电脑都是 64 位版本，但是可能会有 32 位版本的电脑，也可以在中间找到对应的版本。对于 Python 版本，可以选择 2.7 版本，也可以选择 3.6 版本，这里推荐使用 Python 3.6 版本。本书中的全部代码都是基于 Python 3.6 版本来写的。如果你使用 2.7 版本，可能会出现一些小的问题。虽然 Python 2.7 是稳定的版本，但是随着 3.6 版本的不断更新，不久之后 2.7 版本肯定会被淘汰掉，版本更迭是未来的趋势。

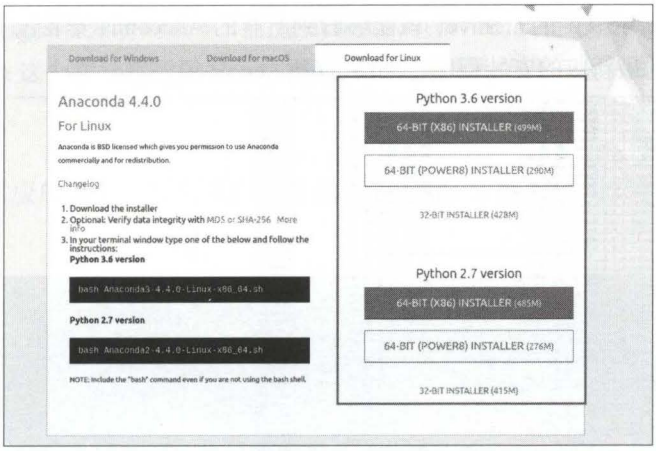


图 2.3 选择 Python 版本

选择好之后单击对应的图标就能够开始下载了，但是 Anaconda 的服务器在国外的，所以速度可能会很慢，我们可以使用清华的镜像来下载，输入网址 https://mirrors.tuna.tsinghua.edu.cn/，进入如图2.4所示的界面。

图 2.4 清华镜像下载

选择框中的 Anaconda 选项，进入如图2.5所示的界面。

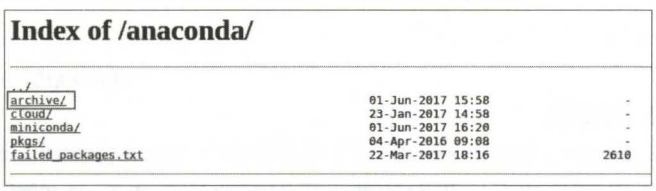


图 2.5 Anaconda 选项

单击图2.5所示框中的 archive，就能够看到所有的 Anaconda 安装包了，滑动鼠标到最下面能够看到最新版的安装程序，如图 2.6所示。

Anaconda3-2.4.1-Linux-x86_64.sh	08-Dec-2015 21:00	283797156
Anaconda3-2.4.1-MacOSX-x86_64.pkg	08-Dec-2015 21:00	259674929
Anaconda3-2.4.1-MacOSX-x86_64.sh	08-Dec-2015 21:00	224248817
Anaconda3-2.4.1-Windows-x86.exe	08-Dec-2015 21:00	313632120
Anaconda3-2.4.1-Windows-x86_64.exe	08-Dec-2015 21:00	381329960
Anaconda3-2.5.0-Linux-x86.sh	03-Feb-2016 21:42	358634167
Anaconda3-2.5.0-Linux-x86_64.sh	03-Feb-2016 21:41	414038933
Anaconda3-2.5.0-MacOSX-x86_64.pkg	03-Feb-2016 21:56	387740293
Anaconda3-2.5.0-MacOSX-x86_64.sh	03-Feb-2016 21:41	333727463
Anaconda3-2.5.0-Windows-x86.exe	03-Feb-2016 21:46	310656008
Anaconda3-2.5.0-Windows-x86_64.exe	03-Feb-2016 21:47	378634984
Anaconda3-4.0.0-Linux-x86.sh	29-Mar-2016 16:15	353266156
Anaconda3-4.0.0-Linux-x86_64.sh	29-Mar-2016 16:15	417790602
Anaconda3-4.0.0-MacOSX-x86_64.pkg	29-Mar-2016 16:16	358139390
Anaconda3-4.0.0-MacOSX-x86_64.sh	29-Mar-2016 16:16	306950558
Anaconda3-4.0.0-Windows-x86.exe	29-Mar-2016 16:16	290840248
Anaconda3-4.0.0-Windows-x86_64.exe	29-Mar-2016 16:16	362171448
Anaconda3-4.1.0-Linux-x86.sh	28-Jun-2016 16:28	344388821
Anaconda3-4.1.0-Linux-x86_64.sh	28-Jun-2016 16:28	424649787
Anaconda3-4.1.0-MacOSX-x86_64.pkg	28-Jun-2016 16:28	363587059
Anaconda3-4.1.0-MacOSX-x86_64.sh	28-Jun-2016 16:28	312081344
Anaconda3-4.1.0-Windows-x86.exe	28-Jun-2016 16:28	306784184
Anaconda3-4.1.0-Windows-x86_64.exe	28-Jun-2016 16:28	368509992
Anaconda3-4.1.1-Linux-x86.sh	08-Jul-2016 16:20	345064389
Anaconda3-4.1.1-Linux-x86_64.sh	08-Jul-2016 16:20	425991075
Anaconda3-4.1.1-MacOSX-x86_64.pkg	08-Jul-2016 16:21	364773025
Anaconda3-4.1.1-MacOSX-x86_64.sh	08-Jul-2016 16:21	313217912
Anaconda3-4.1.1-Windows-x86.exe	08-Jul-2016 16:21	308116424
Anaconda3-4.1.1-Windows-x86_64.exe	08-Jul-2016 16:21	370055720
Anaconda3-4.2.0-Linux-x86.sh	27-Sep-2016 20:50	392066694
Anaconda3-4.2.0-Linux-x86_64.sh	27-Sep-2016 20:50	478051940
Anaconda3-4.2.0-MacOSX-x86_64.pkg	18-Oct-2016 00:33	426843288
Anaconda3-4.2.0-MacOSX-x86_64.sh	27-Sep-2016 20:50	366497043
Anaconda3-4.2.0-Windows-x86.exe	27-Sep-2016 20:56	349560232
Anaconda3-4.2.0-Windows-x86_64.exe	27-Sep-2016 20:57	418431584
Anaconda3-4.3.0-Linux-x86.sh	27-Jan-2017 20:14	417717702
Anaconda3-4.3.0-Linux-x86_64.sh	27-Jan-2017 20:15	496412001
Anaconda3-4.3.0-MacOSX-x86_64.pkg	27-Jan-2017 20:26	443649282
Anaconda3-4.3.0-MacOSX-x86_64.sh	27-Jan-2017 20:26	388197988
Anaconda3-4.3.0-Windows-x86.exe	27-Jan-2017 20:18	364059456
Anaconda3-4.3.0-Windows-x86_64.exe	27-Jan-2017 20:19	441681320
Anaconda3-4.3.0.1-Windows-x86.exe	03-Feb-2017 16:33	364057456
Anaconda3-4.3.0.1-Windows-x86_64.exe	03-Feb-2017 16:33	441680784
Anaconda3-4.3.1-Linux-x86.sh	10-Mar-2017 17:56	418059792
Anaconda3-4.3.1-Linux-x86_64.sh	10-Mar-2017 17:56	497343851
Anaconda3-4.3.1-MacOSX-x86_64.pkg	10-Mar-2017 17:56	444666396
Anaconda3-4.3.1-MacOSX-x86_64.sh	10-Mar-2017 17:57	381078558
Anaconda3-4.3.1-Windows-x86.exe	10-Mar-2017 17:59	365005040
Anaconda3-4.3.1-Windows-x86_64.exe	10-Mar-2017 17:59	442630816
Anaconda3-4.4.0-Linux-ppc64le.sh	30-May-2017 19:23	304862316
Anaconda3-4.4.0-Linux-x86.sh	30-May-2017 19:24	449473324
Anaconda3-4.4.0-Linux-x86_64.sh	30-May-2017 19:24	523283080
Anaconda3-4.4.0-MacOSX-x86_64.pkg	30-May-2017 19:25	464033256
Anaconda3-4.4.0-MacOSX-x86_64.sh	30-May-2017 19:26	398987650
Anaconda3-4.4.0-Windows-x86.exe	30-May-2017 19:26	379794600
Anaconda3-4.4.0-Windows-x86_64.exe	30-May-2017 19:27	458893576

图 2.6　所有 Anaconda 安装包

选择图 2.6 中框中对应系统的版本就能够下载最新的 Anaconda 了，等待下载完成，就可以开始安装了。之后如果有需要的包而 Anaconda 没有的话，可以通过 pip 或者 conda 来安装，比如要安装 numpy，就在终端里面输入命令 pip install numpy 或者 conda install numpy 即可，像 numpy 这样常用的包 Anaconda 里面是自带的。如果需要安装很多包，你会发现 conda 下载的速度经常很慢，这也是因为 Anaconda 的服务器在国外。这时我们可以用刚才的办法，使用清华的镜像，因为清华的镜像里面有 Anaconda 仓库的镜像，只需要将其加入 conda 的配置即可，可以通过在终端中输入以下命令来实现：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/
pkgs/free/
```

这样我们就将清华镜像加入到了 Anaconda 下载的频道中，如果实在遇到什么问题，可以自己动手，去百度上面搜索，一般都能搜到解决办法。

2.3.3　PyTorch 的安装

安装完 Python 环境之后，我们就进入了深度学习库 PyTorch 的安装。首先进入 PyTorch 的官方网站，输入网址 http://pytorch.org，可以进入如图2.7所示的界面。

在框中可以选择要安装的版本，注意如果你的电脑没有支持的显卡可以进行 GPU 加速，那么 Cuda 这个部分就选择 None，然后在终端中运行左下角框中的语句，这样就可以顺利安装了。

如果你有对应的显卡，那么我们需要先安装 Cuda 才能够使用安装 GPU 版本的 PyTorch。

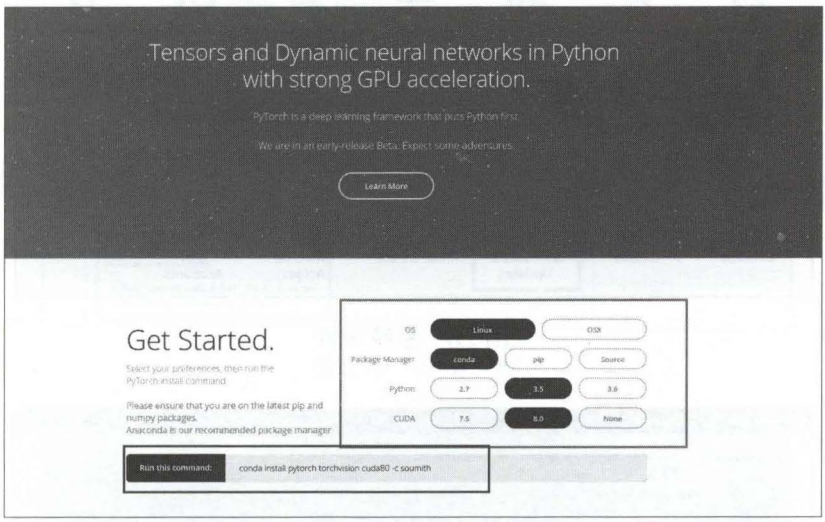


图 2.7 PyTorch 官网首页

对于 GPU 版本的安装，会稍微复杂一点，大概需要以下几个步骤：

（1）安装 Nvidia Cuda；

（2）安装 CuDNN；

（3）安装 GPU 版本 PyTorch；

（4）测试。

（1）安装 Nvidia Cuda

首先需要确认你的电脑显卡已经安装好了驱动并且是支持 Cuda 的，对于 Linux，显卡驱动的安装非常简单，进入系统设置，选择图2.8中框中的系统设置。然后就可以看到如图2.9所示的界面。

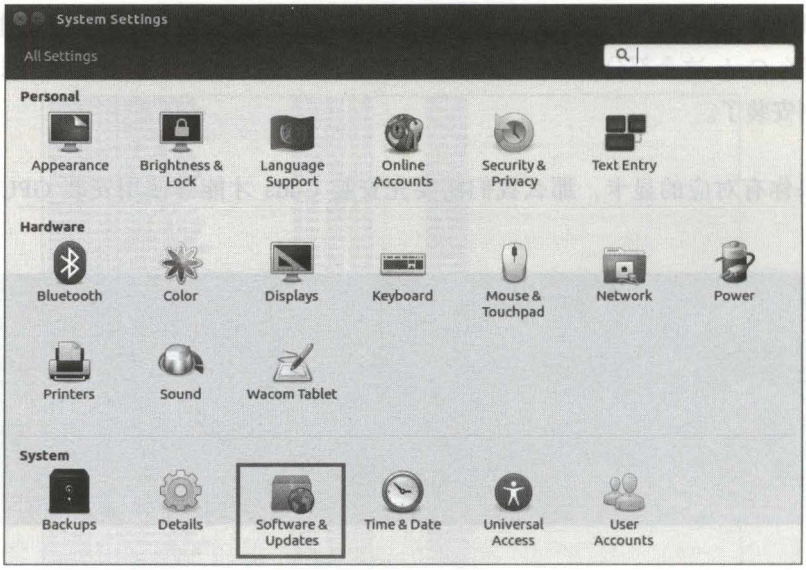


图 2.8　系统设置

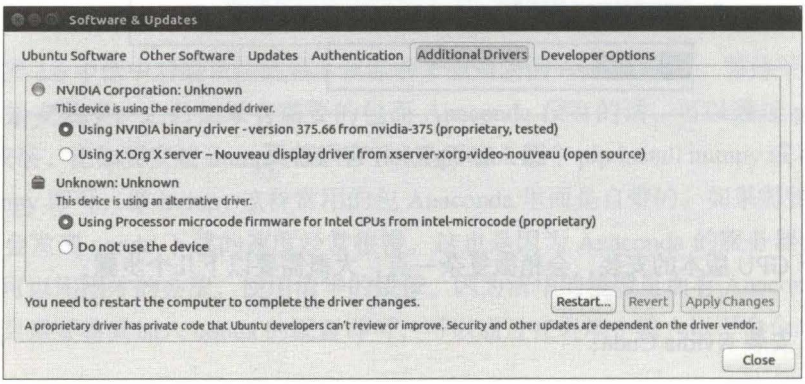


图 2.9　选择显卡驱动

然后在里面选择相应提供的显卡驱动等待安装完成即可。

接着需要安装 Nvidia Cuda，首先进入 https://developer.nvidia.com/cuda-downloads 这个网址，然后按着如图2.10所示的界面选择下载按钮就可以等待下载了。

下载完成后进入到下载文件的目录，接着按照图 2.10 所示的框命令安装就可以了。

然后进入命令界面跟着指示执行即可，但是要注意图 2.11所示的这两个地方。

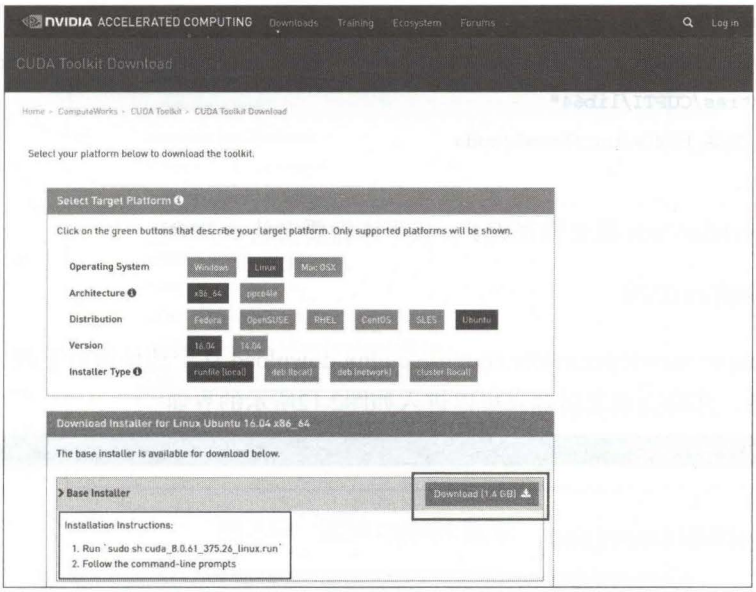


图 2.10 开始下载

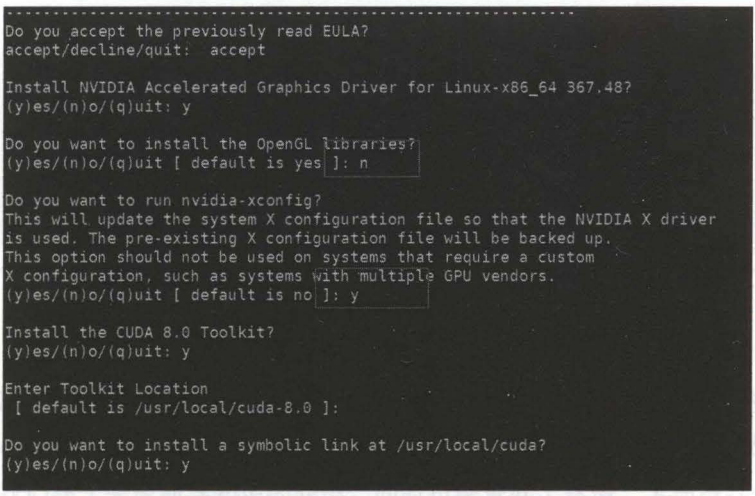


图 2.11 命令界面

一定不要安装 Open GL，这个地方要选 n，否则有可能在安装完成后无法启动图形化桌面，之后重新启动就可以了。

接着添加环境变量，在根目录下打开终端，按照下面指示进行输入：

```
1 $ sudo vim ~/.bash_profile
2
```

```
3 在打开的文本末尾加入：
4 export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/usr/local/
  cuda/extras/CUPTI/lib64"
5 export CUDA_HOME=/usr/local/cuda
```

这样 Nvidia Cuda 就安装完成了，接下来需要安装 cuDNN。

（2）安装 cuDNN

进入 https://developer.nvidia.com/rdp/cudnn-download 这个地址进行下载，下载前需要注册登录，注册完成之后登录可以进入如图2.12所示的界面。

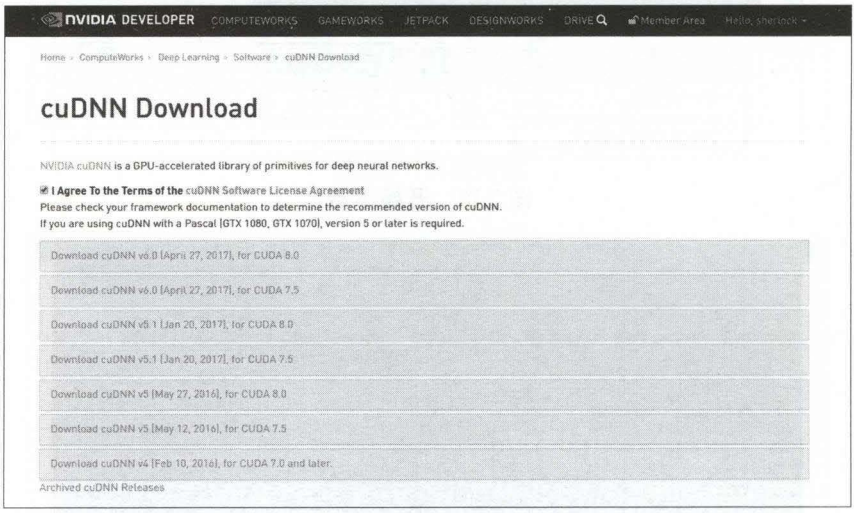


图 2.12　下载 cuDNN

然后选择下载 cuDNN v5.1(Jan 20, 2017), for CUDA 8.0，接着选择如图 2.13所示框中的部分进行下载。

下载下来的是 deb 包，然后运行 dpkg -i 就可以安装了。

（3）GPU 版 PyTorch

安装 GPU 版 PyTorch 跟 CPU 版本差不多，也是先进入官网，只不过最后一行选择 CUDA 8.0，按照这命令行说明就可以安装了。

（4）测试

打开终端，输入 python 进入到 python 界面，然后按照图 2.14输入。

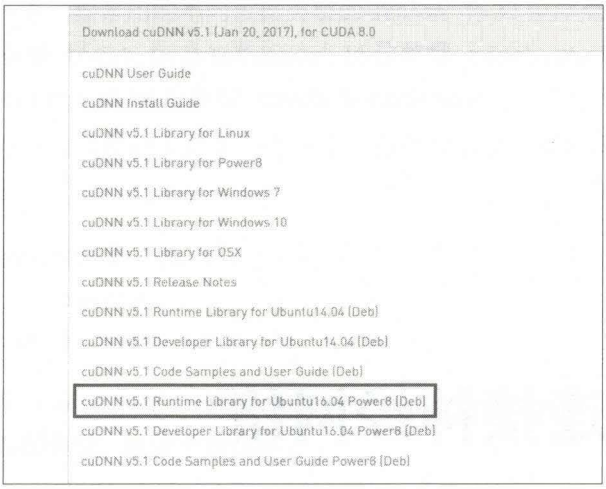


图 2.13 选择 cuDNN 版本

图 2.14 终端界面

如果最后一行出来的结果如图 2.14 所示的是 True，那么恭喜你 GPU 版本的 PyTorch 安装成功了。

到这里我们终于完成前导知识的介绍，同时也配置好了环境，下面就让我们一起进入 PyTorch 的世界。

第 3 章

多层全连接神经网络

深度学习的前身便是多层全连接神经网络，神经网络领域最开始主要是用来模拟人脑神经元系统，但是随后逐渐发展成了一项机器学习技术。多层全连接神经网络是现在深度学习中各种复杂网络的基础，了解它能够帮助我们更好地学习之后的内容。

这一章我们将先从 PyTorch 基础入手，介绍 PyTorch 的处理对象、运算操作、自动求导，以及数据处理方法，接着从线性模型开始进入机器学习的内容，然后由 Logistic 回归引入分类问题，接着介绍多层全连接神经网络、反向传播算法、各种基于梯度的优化算法、数据预处理和训练技巧，最后用 PyTorch 实现多层全连接神经网络。

3.1 热身：PyTorch 基础

首先我们会介绍一下 PyTorch 里面的一些基础知识，有了这些基础知识之后我们才能做出更多复杂的变形。

3.1.1 Tensor（张量）

PyTorch 里面处理的最基本的操作对象就是 Tensor，Tensor 是张量的英文，表示的是一个多维的矩阵，比如零维就是一个点，一维就是向量，二维就是一般的矩阵，多维就相当于一个多维的数组，这和 numpy 是对应的，而且 PyTorch 的 Tensor 可以和 numpy 的 ndarray 相互转换，唯一不同的是 PyTorch 可以在 GPU 上运行，而 numpy 的 ndarray 只能在 CPU 上运行。

我们先介绍一下一些常用的不同数据类型的 Tensor，有 32 位浮点型 `torch.FloatTensor`、64 位浮点型 `torch.DoubleTensor`、16 位整型 `torch.ShortTensor`、32 位整型 `torch.IntTensor` 和 64 位整型 `torch.LongTensor`。

我们可以通过下面这样的方式来定义一个三行两列给定元素的矩阵，并且显示出矩阵的元素和大小：

```
a = torch.Tensor([[2, 3], [4, 8], [7, 9]])
print('a is: {}'.format(a))
print('a size is {}'.format(a.size()))  # a.size() = 3, 2
```

需要注意的是 `torch.Tensor` 默认的是 `torch.FloatTensor` 数据类型，也可以定义我们想要的数据类型，就像下面这样：

```
b = torch.LongTensor([[2, 3], [4, 8], [7, 9]])
print('b is : {}'.format(b))
```

当然也可以创建一个全是 0 的空 Tensor 或者取一个正态分布作为随机初始值：

```
c = torch.zeros((3, 2))
print('zero tensor: {}'.format(c))

d = torch.randn((3, 2))
print('normal randon is : {}'.format(d))
```

我们也可以像 numpy 一样通过索引的方式取得其中的元素，同时也可以改变它的值，比如将 a 的第一行第二列改变为 100。

```
a[0, 1] = 100
print('changed a is: {}'.format(a))
```

除此之外，还可以在 Tensor 与 numpy.ndarray 之间相互转换：

```
numpy_b = b.numpy()
print('conver to numpy is \n {}'.format(numpy_b))

e = np.array([[2, 3], [4, 5]])
torch_e = torch.from_numpy(e)
print('from numpy to torch.Tensor is {}'.format(torch_e))
f_torch_e = torch_e.float()
print('change data type to float tensor: {}'.format(f_torch_e))
```

通过简单的 `b.numpy()`，就能将 b 转换为 numpy 数据类型，同时使用 `torch.from_num py()` 就能将 numpy 转换为 tensor，如果需要更改 tensor 的数据类型，只需要在转换后的 tensor 后面加上你需要的类型，比如想将 a 的类型转换成 float，只需 `a.float()` 就可以了。

如果你的电脑支持 GPU 加速，还可以将 Tensor 放到 GPU 上。

首先通过 `torch.cuda.is_available()` 判断一下是否支持 GPU，如果想把 tensor a 放到 GPU 上，只需 `a.cuda()` 就能够将 tensor a 放到 GPU 上了。

```
if torch.cuda.is_available():
    a_cuda = a.cuda()
    print(a_cuda)
```

3.1.2 Variable（变量）

接着要讲的一个概念就是 Variable，也就是变量，这个在 numpy 里面就没有了，是神经网络计算图里特有的一个概念，就是 Variable 提供了自动求导的功能，之前如果了解过 Tensorflow 的读者应该清楚神经网络在做运算的时候需要先构造一个计算图谱，然后在里面进行前向传播和反向传播。

Variable 和 Tensor 本质上没有区别，不过 Variable 会被放入一个计算图中，然后进行前向传播，反向传播，自动求导。

首先 Variable 是在 `torch.autograd.Variable` 中，要将一个 tensor 变成 Variable 也非常简单，比如想让一个 tensor a 变成 Variable，只需要 `Variable(a)` 就可以了。我们可以通过图 3.1了解到 Variable 的属性。

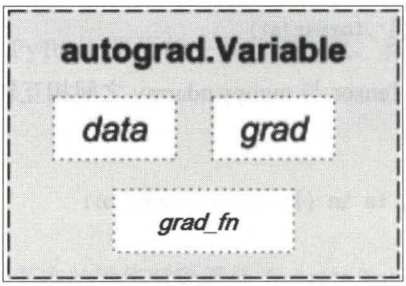


图 3.1　结构图

从图3.1中可以看出 Variable 有三个比较重要的组成属性：data，grad 和 grad_fn。通过 data 可以取出 Variable 里面的 tensor 数值，grad_fn 表示的是得到这个 Variable 的操

作，比如通过加减还是乘除来得到的，最后 grad 是这个 Variabel 的反向传播梯度，下面通过例子来具体说明一下：

```
# Create Variable
x = Variable(torch.Tensor([1]), requires_grad=True)
w = Variable(torch.Tensor([2]), requires_grad=True)
b = Variable(torch.Tensor([3]), requires_grad=True)

# Build a computational graph.
y = w * x + b    # y = 2 * x + 3

# Compute gradients
y.backward()  # same as y.backward(torch.FloatTensor([1]))
# Print out the gradients.
print(x.grad)    # x.grad = 2
print(w.grad)    # w.grad = 1
print(b.grad)    # b.grad = 1
```

构建 Variable，要注意得传入一个参数 `requires_grad=True`，这个参数表示是否对这个变量求梯度，默认的是 False，也就是不对这个变量求梯度，这里我们希望得到这些变量的梯度，所以需要传入这个参数。

从上面的代码中，我们注意到了一行 `y.backward()`，这一行代码就是所谓的自动求导，这其实等价于 `y.backward(torch.FloatTensor([1]))`，只不过对于标量求导里面的参数就可以不写了，自动求导不需要你再去明确地写明哪个函数对哪个函数求导，直接通过这行代码就能对所有的需要梯度的变量进行求导，得到它们的梯度，然后通过 `x.grad` 可以得到 x 的梯度。

上面是标量的求导，同时也可以做矩阵求导，比如：

```
x = torch.randn(3)
x = Variable(x, requires_grad=True)

y = x * 2
print(y)

y.backward(torch.FloatTensor([1, 0.1, 0.01]))
print(x.grad)
```

相当于给出了一个三维向量去做运算，这时候得到的结果 y 就是一个向量，这里对这个向量求导就不能直接写成 `y.backward()`，这样程序是会报错的。这个时候需要传入参数声明，比如 `y.backward(torch.FloatTensor([1, 1, 1]))`，这样得到的结果就是它们每个分量的梯度，或者可以传入 `y.backward(torch.FloatTensor([1, 0.1, 0.01]))`，这样得到的梯度就是它们原本的梯度分别乘上 1，0.1 和 0.01。

3.1.3 Dataset（数据集）

在处理任何机器学习问题之前都需要数据读取，并进行预处理。PyTorch 提供了很多工具使得数据的读取和预处理变得很容易。

`torch.utils.data.Dataset` 是代表这一数据的抽象类，你可以自己定义你的数据类继承和重写这个抽象类，非常简单，只需要定义`__len__`和`__getitem__`这两个函数：

```
class myDataset(Dataset):
    def __init__(self, csv_file, txt_file, root_dir, other_file):
        self.csv_data = pd.read_csv(csv_file)
        with open(txt_file, 'r') as f:
            data_list = f.readlines()
        self.txt_data = data_list
        self.root_dir = root_dir

    def __len__(self):
        return len(self.csv_data)

    def __getitem__(self, idx):
        data = (self.csv_data[idx], self.txt_data[idx])
        return data
```

通过上面的方式，可以定义我们需要的数据类，可以通过迭代的方式来取得每一个数据，但是这样很难实现取 batch，shuffle 或者是多线程去读取数据，所以 PyTorch 中提供了一个简单的办法来做这个事情，通过 `torch.utils.data.DataLoader` 来定义一个新的迭代器，如下：

```
dataiter = DataLoader(myDataset, batch_size=32, shuffle=True,
                      collate_fn=default_collate)
```

里面的参数都特别清楚，只有 collate_fn 是表示如何取样本的，我们可以定义自己的函数来准确地实现想要的功能，默认的函数在一般情况下都是可以使用的。

另外在 torchvision 这个包中还有一个更高级的有关于计算机视觉的数据读取类：ImageFolder，主要功能是处理图片，且要求图片是下面这种存放形式：

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/asd.png
root/cat/zxc.png
```

之后这样来调用这个类：

```
dset = ImageFolder(root='root_path', transform=None,loader=default_loader)
```

其中的 root 需要是根目录，在这个目录下有几个文件夹，每个文件夹表示一个类别：transform 和 target_transform 是图片增强，之后我们会详细讲；loader 是图片读取的办法，因为我们读取的是图片的名字，然后通过 loader 将图片转换成我们需要的图片类型进入神经网络。

3.1.4 nn.Module（模组）

在 PyTorch 里面编写神经网络，所有的层结构和损失函数都来自于 torch.nn，所有的模型构建都是从这个基类 nn.Module 继承的，于是有了下面这个模板。

```
class net_name(nn.Module):
    def __init__(self, other_arguments):
        super(net_name, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size)
        # other network layer

    def forward(self, x):
        x = self.conv1(x)
        return x
```

这样就建立了一个计算图，并且这个结构可以复用多次，每次调用就相当于用该计算图定义的相同参数做一次前向传播，这得益于 PyTorch 的自动求导功能，所以我们不需要自己编写反向传播，而所有的网络层都是由 nn 这个包得到的，比如线性层 `nn.Linear`，等之后使用的时候我们可以详细地介绍每一种网络对应的结构，以及如何调用。

定义完模型之后，我们需要通过 nn 这个包来定义损失函数。常见的损失函数都已经定义在了 nn 中，比如均方误差、多分类的交叉熵，以及二分类的交叉熵，等等，调用这些已经定义好的损失函数也很简单：

```
criterion = nn.CrossEntropyLoss()
loss = criterion(output, target)
```

这样就能求得我们的输出和真实目标之间的损失函数了。

3.1.5 torch.optim（优化）

在机器学习或者深度学习中，我们需要通过修改参数使得损失函数最小化（或最大化），优化算法就是一种调整模型参数更新的策略。

优化算法分为两大类。

1. 一阶优化算法

这种算法使用各个参数的梯度值来更新参数，最常用的一阶优化算法是梯度下降。所谓的梯度就是导数的多变量表达式，函数的梯度形成了一个向量场，同时也是一个方向，这个方向上方向导数最大，且等于梯度。梯度下降的功能是通过寻找最小值，控制方差，更新模型参数，最终使模型收敛，网络的参数更新公式是：

$$\theta = \theta - \eta \times \frac{\partial J(\theta)}{\partial \theta} \tag{3.1}$$

其中 η 是学习率，$\frac{\partial J(\theta)}{\partial \theta}$ 是函数的梯度，我们可以通过图 3.2形象地说明一下该方法。

这是深度学习里面最常用的优化方法，我们之后会详细讲解它的各种变式。

2. 二阶优化算法

二阶优化算法使用了二阶导数（也叫做 Hessian 方法）来最小化或最大化损失函数，主要基于牛顿法，但是由于二阶导数的计算成本很高，所以这种方法并没有广泛使用。`torch.optim` 是一个实现各种优化算法的包，大多数常见的算法都能够直接通过这个包来调用，比如随机梯度下降，以及添加动量的随机梯度下降，自适应学习率等。

在调用的时候将需要优化的参数传入，这些参数都必须是 Variable，然后传入一些基本的设定，比如学习率和动量等。

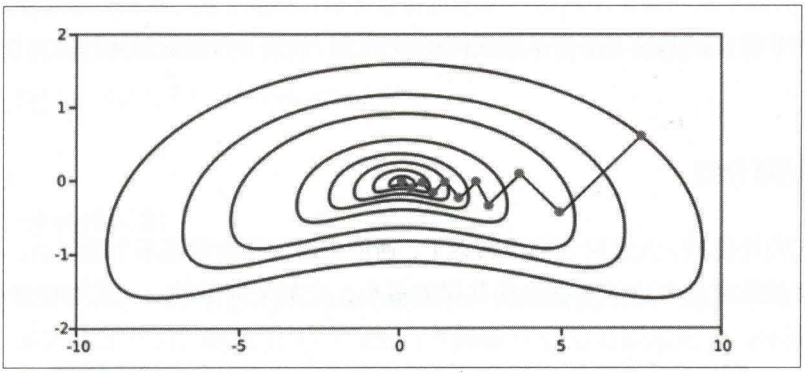


图 3.2 一阶优化算法

下面举一个例子：

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

这样我们就将模型的参数作为需要更新的参数传入优化器，设定学习率是 0.01，动量是 0.9 的随机梯度下降，在优化之前需要先将梯度归零，即 `optimizer.zeros()`，然后通过 `loss.backward()` 反向传播，自动求导得到每个参数的梯度，最后只需要 `optimizer.step()` 就可以通过梯度做一步参数更新。

3.1.6 模型的保存和加载

在 PyTorch 里面使用 torch.save 来保存模型的结构和参数，有两种保存方式：

（1）保存整个模型的结构信息和参数信息，保存的对象是模型 model；

（2）保存模型的参数，保存的对象是模型的状态 `model.state_dict()`。

可以这样保存，save 的第一个参数是保存对象，第二个参数是保存路径及名称：

```
torch.save(model, './model.pth')
torch.save(model.state_dict(), './model_state.pth')
```

加载模型有两种方式对应于保存模型的方式：

（1）加载完整的模型结构和参数信息，使用 `load_model = torch.load('model.pth')`，在网络较大的时候加载的时间比较长，同时存储空间也比较大；

（2）加载模型参数信息，需要先导入模型的结构，然后通过 `model.load_state_dic (torch.load('model_state.pth'))` 来导入。

3.2 线性模型

这一节将从机器学习最简单的线性模型入手，看看 PyTorch 如何解决这个问题。

3.2.1 问题介绍

说起线性模型，大家对它都很熟悉了，通俗来讲就是给定很多个数据点，希望能够找到一个函数来拟合这些数据点使其误差最小，比如最简单的一元线性模型就可以用图3.3来表示。

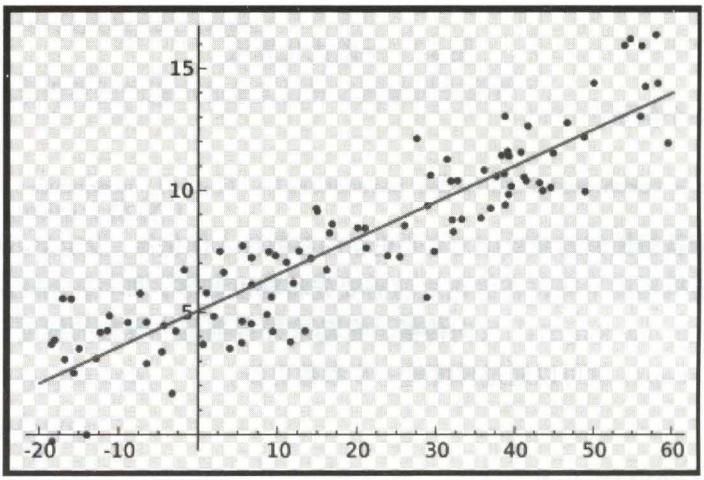


图 3.3 一元线性模型

图3.3给出了一系列的点，找一条直线，使得直线尽可能与这些点接近，也就是这些点到直线的距离之和尽可能小。用数学语言来严格表达，即给定由 d 个属性描述的示例 $x = (x_1, x_2, x_3, \cdots, x_d)$，其中 x_i 表示 x 在第 i 个属性上面的取值，线性模型就是试图学习一个通过属性的线性组合来进行预测的函数，即

$$f(x) = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b \tag{3.2}$$

一般可以用向量的形式来表达：

$$f(x) = w^T x + b \tag{3.3}$$

其中 $w = (w_1, w_2, \cdots, w_d)$，$w$ 和 b 都是需要学习的参数，模型通过不断地调整 w 和 b，

最后就能够得到一个最优的模型。

线性模型形式简单、易于建模，却孕育着机器学习领域中重要的基本思想，同时线性模型还具有特别好的解释性，因为权重的大小就直接可以表示这个属性的重要程度。

首先我们从最简单的一维线性回归入手。

3.2.2 一维线性回归

给定数据集 $D = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \cdots, (x_m, y_m)\}$，线性回归希望能够优化出一个好的函数 $f(x)$，使得 $f(x_i) = wx_i + b$ 能够与 y_i 尽可能接近。

如何才能学习到参数 w 和 b 呢？很简单，只需要确定如何衡量 $f(x)$ 与 y 之间的差别，就像之前讲的一样，可以用它们之间的距离差异 $Loss = \sum_{i=1}^{m}(f(x_i) - y_i)^2$ 来衡量误差，取平方是因为距离有正有负，我们希望能够将它们全部变成正的。这也就是著名的均方误差，要做的事情就是希望能够找到 w^* 和 b^*，使得

$$(w^*, b^*) = \arg\min_{w,b} \sum_{i=1}^{m}(f(x_i) - y_i)^2 \tag{3.4}$$

$$= \arg\min_{w,b} \sum_{i=1}^{m}(y_i - wx_i - b)^2 \tag{3.5}$$

均方误差非常直观，也有着很好的几何意义，对应了常用的欧几里得距离，基于均方误差最小化来进行模型求解的办法也称为“最小二乘法”。

求解办法其实非常简单，如果求这个连续函数的最小值，那么只需要求它的偏导数，让它的偏导数等于 0 来估计它的参数，即：

$$\frac{\partial Loss_{(w,b)}}{\partial w} = 2(w\sum_{i=1}^{m} x_i^2 - \sum_{i=1}^{m}(y_i - b)x_i) = 0 \tag{3.6}$$

$$\frac{\partial Loss_{(w,b)}}{\partial b} = 2(mb - \sum_{i=1}^{m}(y_i - wx_i)) = 0 \tag{3.7}$$

通过求解式 (3.6) 和式 (3.7)，我们就可以得到 w 和 b 的最优解：

$$w = \frac{\sum_{i=1}^{m} y_i(x_i - \bar{x})}{\sum_{i=1}^{m} x_i^2 - \frac{1}{m}(\sum_{i=1}^{m} x_i)^2} \tag{3.8}$$

$$b = \frac{1}{m}\sum_{i=1}^{m}(y_i - wx_i) \tag{3.9}$$

其中 $\bar{x} = \frac{1}{m}\sum_{i=1}^{m} x_i$ 即为 x 的均值。

3.2.3 多维线性回归

更一般的情况是多维线性回归，比如像前文描述的，我们有 d 个属性，试图学得最优的函数 $f(x)$：

$$f(x_i) = w^T x_i + b \tag{3.10}$$

使得 $\sum_{i=1}^{m}(f(x_i) - y_i)^2$ 最小，这也称为“多元线性回归”，同样可以用最小二乘法对 w 和 b 进行估计，为了方便计算，可以将 w 和 d 写进同一个矩阵，将数据集 D 表示成一个 $m \times (d+1)$ 的矩阵 X，每行前面 d 个元素表示 d 个属性值，最后一个元素设为 1，即

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} & 1 \\ x_{21} & x_{22} & \cdots & x_{2d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{md} & 1 \end{pmatrix} = \begin{pmatrix} x_1^T & 1 \\ x_2^T & 1 \\ \vdots & \vdots \\ x_m^T & 1 \end{pmatrix}$$

将目标 y 也写乘向量的形式 $y = (y_1, y_2, \cdots, y_m)$，那么我们就能够得到：

$$w^* = \arg\min_w (y - Xw)^T (y - Xw) \tag{3.11}$$

同样对其求导，令它等于 0。

$$\frac{\partial Loss_w}{\partial w} = 2X^T(Xw - y) = 0 \tag{3.12}$$

因为上面涉及了矩阵的逆运算，所以需要 $X^T X$ 是一个满秩矩阵或者正定矩阵，那么我们可以得到：

$$w^* = (X^T X)^{-1} X^T y \tag{3.13}$$

所以线性回归模型可以写成：

$$f(x_i) = x_i^T (X^T X)^{-1} X^T y \tag{3.14}$$

然而在现实任务中，X^TX 往往不是满秩矩阵，就算是满秩矩阵，求解逆的过程也比较慢，所以我们一般可以使用梯度下降法去求解这个最小二乘问题。

3.2.4 一维线性回归的代码实现

讲了这么多原理，下面用 PyTorch 来求解一下一维线性回归问题。

首先我们随便给出一些点：

```
x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168],
                    [9.779], [6.182], [7.59], [2.167], [7.042],
                    [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)

y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573],
                    [3.366], [2.596], [2.53], [1.221], [2.827],
                    [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)
```

通过 matplotlib 画出来就是这个样子，如图3.4所示。

我们想要做的事情就是找一条直线去逼近这些点，也就是希望这条直线离这些点的距离之和最小，先将 numpy.array 转换成 Tensor，因为 PyTorch 里面的处理单元是 Tensor，按上一章讲的方法，这就特别简单了：

```
x_train = torch.from_numpy(x_train)
y_train = torch.from_numpy(y_train)
```

接着需要建立模型，根据上一节 PyTorch 的基础知识，这样来定义一个简单的模型：

```
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(1, 1)  # input and output is 1 dimension

    def forward(self, x):
        out = self.linear(x)
        return out

if torch.cuda.is_available():
    model = LinearRegression().cuda()
```

```
else:
    model = LinearRegression()
```

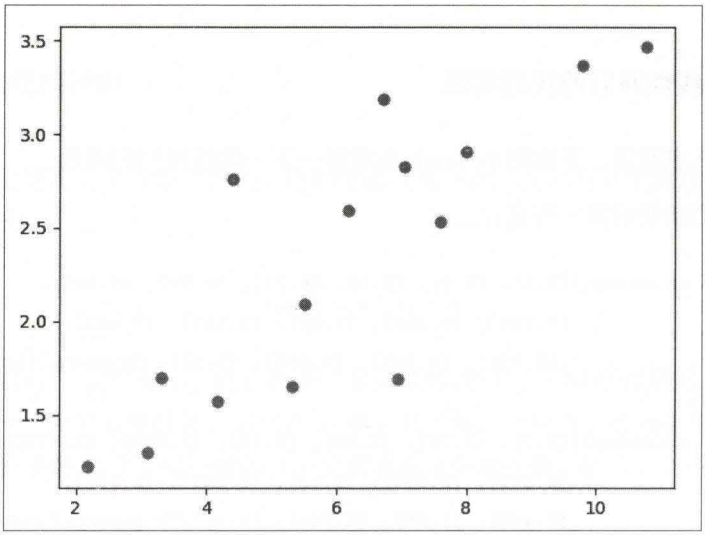


图 3.4　matplotlib 画出的图

这里我们就定义了一个超级简单的模型 $y = wx + b$，输入参数是一维，输出参数也是一维，这就是一条直线，当然这里可以根据你想要的输入输出维度进行更改，我们希望去优化参数 w 和 b 能够使得这条直线尽可能接近这些点，如果支持 GPU 加速，可以通过 `model.cuda()` 将模型放到 GPU 上。

然后定义损失函数和优化函数，这里使用均方误差作为优化函数，使用梯度下降进行优化：

```
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=1e-3)
```

接着就可以开始训练我们的模型了：

```
num_epochs = 1000
for epoch in range(num_epochs):
    if torch.cuda.is_available():
        inputs = Variable(x_train).cuda()
        target = Variable(y_train).cuda()
    else:
        inputs = Variable(x_train)
        target = Variable(y_train)
```

```
9
10      # forward
11      out = model(inputs)
12      loss = criterion(out, target)
13      # backward
14      optimizer.zero_grad()
15      loss.backward()
16      optimizer.step()
17
18      if (epoch+1) % 20 == 0:
19          print('Epoch[{}/{}], loss: {:.6f}'
20                .format(epoch+1, num_epochs, loss.data[0]))
```

定义好我们要跑的 epoch 个数，然后将数据变成 Variable 放入计算图，然后通过 `out=model(inputs)` 得到网络前向传播得到的结果，通过 `loss=criterion(out, target)` 得到损失函数，然后归零梯度，做反向传播和更新参数，特别要注意的是，每次做反向传播之前都要归零梯度，`optimizer.zero_grad()`。不然梯度会累加在一起，造成结果不收敛。在训练的过程中隔一段时间就将损失函数的值打印出来看看，确保我们的模型误差越来越小。注意 `loss.data[0]`，首先 loss 是一个 Variable，所以通过 `loss.data` 可以取出一个 Tensor，再通过 `loss.data[0]` 得到一个 int 或者 float 类型的数据，这样我们才能够打印出相应的数据。

做完训练之后可以预测一下结果。

```
model.eval()
model.cpu()
predict = model(Variable(x_train))
predict = predict.data.numpy()
plt.plot(x_train.numpy(), y_train.numpy(), 'ro', label='Original data')
plt.plot(x_train.numpy(), predict, label='Fitting Line')
plt.show()
```

首先需要通过 `model.eval()` 将模型变成测试模式，这是因为有一些层操作，比如 Dropout 和 BatchNormalization 在训练和测试的时候是不一样的，所以我们需要通过这样一个操作来转换这些不一样的层操作。然后将测试数据放入网络做前向传播得到结果，最后画出的结果如图 3.5所示。

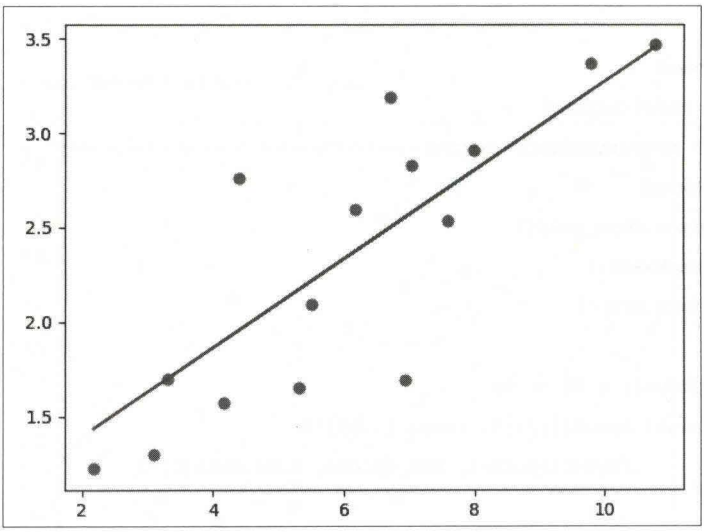


图 3.5　一元回归

这样我们就通过 PyTorch 解决了一个简单的一元回归问题，得到了一条直线去尽可能逼近这些离散的点。

3.2.5　多项式回归

对于一般的线性回归，由于该函数拟合出来的是一条直线，所以精度欠佳，我们可以考虑多项式回归，也就是提高每个属性的次数，而不再是只使用一次去回归目标函数。

原理和之前的线性回归是一样的，只不过这里用的是高次多项式而不是简单的一次线性多项式。

首先给出我们想要拟合的方程：

$$y = 0.9 + 0.5 \times x + 3 \times x^2 + 2.4 \times x^3$$

然后可以设置参数方程：

$$y = b + w_1 \times x + w_2 \times x^2 + w_3 \times x^3$$

我们希望每一个参数都能够学习到和真实参数很接近的结果。

下面来看看如何用 PyTorch 实现这个简单的任务。

首先需要预处理数据，也就是需要将数据变成一个矩阵的形式：

$$X = \begin{pmatrix} x_1 & x_1^2 & x_1^3 \\ x_3 & x_3^2 & x_3^3 \\ \vdots & \ddots & \vdots \\ x_n & x_n^2 & x_n^3 \end{pmatrix}$$

在 PyTorch 里面使用 `torch.cat()` 函数来实现 Tensor 的拼接：

```
def make_features(x):
    """Builds features i.e. a matrix with columns [x, x^2, x^3]."""
    x = x.unsqueeze(1)
    return torch.cat([x ** i for i in range(1, 4)], 1)
```

对于输入的 n 个数据，我们将其扩展成上面矩阵所示的样子。

然后定义好真实的函数：

```
W_target = torch.FloatTensor([0.5, 3, 2.4]).unsqueeze(1)
b_target = torch.FloatTensor([0.9])

def f(x):
    """Approximated function."""
    return x.mm(W_target) + b_target[0]
```

这里的权重已经定义好了，`unsqueeze(1)` 是将原来的 tensor 大小由 3 变成 (3, 1)，`x.mm(W_target)` 表示做矩阵乘法，$f(x)$ 就是每次输入一个 x 得到一个 y 的真实函数。

在进行训练的时候我们需要采样一些点，可以随机生成一些数来得到每次的训练集：

```
def get_batch(batch_size=32):
    """Builds a batch i.e. (x, f(x)) pair."""
    random = torch.randn(batch_size)
    x = make_features(random)
    y = f(x)
    if torch.cuda.is_available():
        return Variable(x).cuda(), Variable(y).cuda()
    else:
        return Variable(x), Variable(y)
```

通过上面这个函数我们每次取 batch_size 这么多个数据点，然后将其转换成矩阵的形式，再把这个值通过函数之后的结果也返回作为真实的目标。

然后可以定义多项式模型：

```
# Define model
class poly_model(nn.Module):
    def __init__(self):
        super(poly_model, self).__init__()
        self.poly = nn.Linear(3, 1)

    def forward(self, x):
        out = self.poly(x)
        return out

if torch.cuda.is_available():
    model = poly_model().cuda()
else:
    model = poly_model()
```

这里的模型输入是 3 维，输出是 1 维，跟之前定义的线性模型只有很小的差别。

然后我们定义损失函数和优化器：

```
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=1e-3)
```

同样使用均方误差来衡量模型的好坏，使用随机梯度下降来优化模型，然后开始训练模型：

```
epoch = 0
while True:
    # Get data
    batch_x, batch_y = get_batch()
    # Forward pass
    output = model(batch_x)
    loss = criterion(output, batch_y)
    print_loss = loss.data[0]
    # Reset gradients
    optimizer.zero_grad()
```

```
    # Backward pass
    loss.backward()
    # update parameters
    optimizer.step()
    epoch += 1
    if print_loss < 1e-3:
        break
```

这里我们希望模型能够不断地优化，直到实现我们设立的条件，取出的 32 个点的均方误差能够小于 0.001。

运行程序可以得到如图3.6所示的结果。

```
Loss: 0.000859 after 2710 batches
==> Learned function:    y = 0.91 + 0.44*x + 2.99*x^2 + 2.41*x^3
==> Actual function:     y = 0.90 + 0.50*x + 3.00*x^2 + 2.40*x^3
```

图 3.6 程序运行结果

将真实函数的数据点和拟合的多项式画在同一张图上，我们可以得到如图3.7所示的结果。

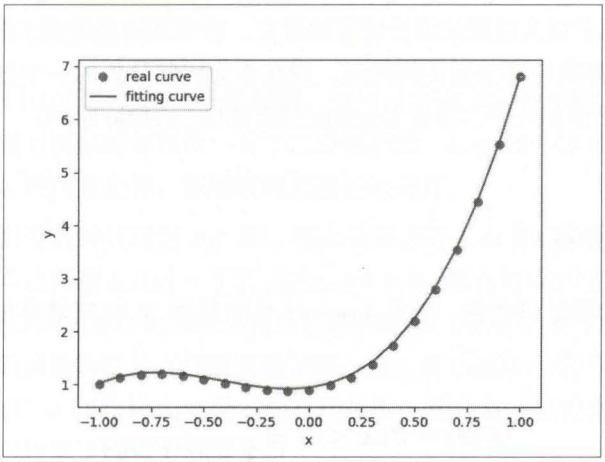


图 3.7 多项式回归

从两个结果来看，我们已经很接近真实的函数了。现实世界中很多问题都不是简单的线性回归，涉及很多复杂的非线性模型，而线性模型是机器学习中最重要的模型之一，它的统计思想及其非常直观的解释性仍然可以给我们一些启发。

3.3 分类问题

下面这节我们将开始介绍机器学习领域里面的另一个问题：分类问题。

3.3.1 问题介绍

机器学习中的监督学习主要分为回归问题和分类问题，我们之前已经讲过回归问题了，它希望预测的结果是连续的，那么分类问题所预测的结果就是离散的类别。这时输入变量可以是离散的，也可以是连续的，而监督学习从数据中学习一个分类模型或者分类决策函数，它被称为分类器（classifier）。分类器对新的输入进行输出预测，这个过程即称为分类（classification）。例如，判断邮件是否为垃圾邮件，医生判断病人是否生病，或者预测明天天气是否下雨等。同时分类问题中包括有二分类和多分类问题，我们下面先讲一下最著名的二分类算法——Logistic 回归。首先从 Logistic 回归的起源说起。

3.3.2 Logistic 起源

Logistic 起源于对人口数量增长情况的研究，后来又被应用到了对于微生物生长情况的研究，以及解决经济学相关的问题，现在作为回归分析的一个分支来处理分类问题，先从 Logistic 分布入手，再由 Logistic 分布推出 Logistic 回归。

3.3.3 Logistic 分布

设 X 是连续的随机变量，服从 Logistic 分布是指 X 的积累分布函数和密度函数如下：

$$F(x) = P(X \leq x) = \frac{1}{1 + e^{-(x-\mu)/\gamma}} \tag{3.15}$$

$$f(x) = \frac{e^{-(x-\mu)/\gamma}}{\gamma(1 + e^{-(x-\mu)/\gamma})^2} \tag{3.16}$$

其中 μ 影响中心对称点的位置，γ 越小中心点附近的增长速度越快。下一节会讲到在深度学习中常用的一个非线性变换 Sigmoid 函数是 Logistic 分布函数中 $\gamma = 1, \mu = 0$ 的特殊形式。

(sigmoid)Logistic 函数的表达形式如下：

$$p(x) = \frac{1}{1+e^{-x}} \tag{3.17}$$

其函数图像如图 3.8所示，由于函数很像“S”形，所以该函数又叫 Sigmoid 函数。

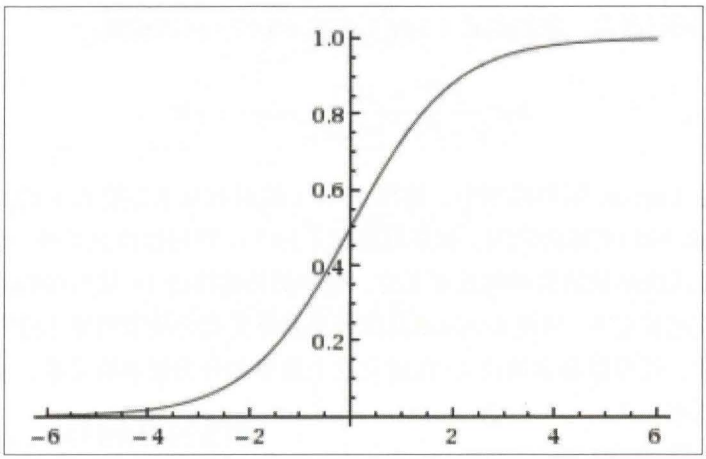


图 3.8 Sigmoid 函数图像

3.3.4 二分类的 Logistic 回归

Logistic 回归不仅可以解决二分类问题，也可以解决多分类问题，但是二分类问题最为常见同时也具有良好的解释性。对于二分类问题，Logistic 回归的目标是希望找到一个区分度足够好的决策边界，能够将两类很好地分开。

假设输入的数据的特征向量 $x \in R^n$，那么决策边界可以表示为 $\sum_{i=1}^{n} w_i x_i + b = 0$；假设存在一个样本点使得 $h_w(x) = \sum_{i=1}^{n} w_i x_i + b > 0$，那么可以判定它的类别是 1；如果 $h_w(x) = \sum_{i=1}^{m} w_i x_i + b < 0$，那么可以判定其类别是 0。这个过程其实是一个感知机的过程，通过决策函数的符号来判断其属于哪一类。而 Logistic 回归要更进一步，通过找到分类概率 $P(Y=1)$ 与输入变量 x 的直接关系，然后通过比较概率值来判断类别，简单来说就是通过计算下面两个概率分布：

$$P(Y=0|x) = \frac{1}{1+e^{w\cdot x+b}} \tag{3.18}$$

$$P(Y=1|x) = \frac{e^{w\cdot x+b}}{1+e^{w\cdot x+b}} \tag{3.19}$$

其中 w 是权重，b 是偏置。现在介绍 Logistic 模型的特点，先引入一个概念：一个事件

发生的几率（odds）是指该事件发生的概率与不发生的概率的比值，比如一个事件发生的概率是 p，那么该事件发生的几率是 $\frac{p}{1-p}$，该事件的对数几率或 logit 函数是：

$$logit(p) = \log \frac{p}{1-p} \tag{3.20}$$

对于 Logistic 回归而言，我们由式（3.16）和式（3.17）可以得到：

$$\log \frac{P(Y=1|x)}{1-P(Y=1|x)} = w \cdot x + b \tag{3.21}$$

这也就是说在 Logistic 回归模型中，输出 $Y=1$ 的对数几率是输入 x 的线性函数，这也就是 Logistic 回归名称的原因。如果观察式（3.17），则可以得到另外一种 Logistic 回归的定义，即线性函数的值越接近正无穷，概率值就越接近 1；线性函数的值越接近负无穷，概率值越接近 0。因此 Logistic 回归的思路是先拟合决策边界（这里的决策边界不局限于线性，还可以是多项式），在建立这个边界和分类概率的关系，从而得到二分类情况下的概率。

3.3.5 模型的参数估计

上面我们简单地介绍了 Logistic 回归模型的建立，下面通过极大似然估计来求出模型中的参数。

对于给定的训练集数据 $T = \{(x_1, y_1), (x_2, y_2), \cdots (x_n, y_n)\}$，其中 $x_i \in R^n, y_i \in \{0, 1\}$，假设 $P(Y=1|x) = \pi(x)$，那么 $P(Y=0|x) = 1 - \pi(x)$，所以似然函数可以有如下的表达：

$$\Pi_{i=1}^{n} [\pi(x_i)]^{y_1} [1 - \pi(x_i)]^{1-y_i} \tag{3.22}$$

取对数之后的对数似然函数为：

$$L(w) = \sum_{i=1}^{n} [y_i \log \pi(x_i) + (1 - y_i) \log(1 - \pi(x_i))] \tag{3.23}$$

$$= \sum_{i=1}^{n} [y_i \log \frac{\pi(x_i)}{1 - \pi(x_i)} + \log(1 - \pi(x_i))] \tag{3.24}$$

$$= \sum_{i=1}^{n} [y_i (w \cdot x_i + b) - \log(1 + e^{w \cdot x_i + b})] \tag{3.25}$$

对 $L(w)$ 求极大值就能够得到 w 的估计值，可以使用最简单的梯度下降法来求得。

用 $L(w)$ 对 W 求导，可以得到：

$$\frac{\partial L(w)}{\partial w} = \sum_{i=1}^{n} y_i x_i - \sum_{i=1}^{n} \frac{e^{w \cdot x_i + b}}{1 + e^{w \cdot x_i + b}} x_i \tag{3.26}$$

$$= \sum_{i=1}^{n} (y_i - logit(w \cdot x_i)) x_i \tag{3.27}$$

$$\frac{\partial L(w)}{\partial b} = \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} \frac{e^{w \cdot x_i + b}}{1 + e^{w \cdot x_i + b}} \tag{3.28}$$

$$= \sum_{i=1}^{n} (y_i - logit(w \cdot x_i)) \tag{3.29}$$

求出梯度之后就可以使用迭代的梯度下降来求解。

3.3.6 Logistic 回归的代码实现

首先我们打开 txt 文件，可以看到数据存放的方式，如图3.9所示。

```
34.62365962451697,78.0246928153624,0
30.28671076822607,43.89499752400101,0
35.84740876993872,72.90219802708364,0
60.18259938620976,86.30855209546826,1
79.0327360507101,75.3443764369103,1
45.08327747668339,56.3163717815305,0
61.10666453684766,96.51142588489624,1
75.02474556738889,46.55401354116538,1
76.09878670226257,87.42056971926803,1
84.43281996120035,43.53339331072109,1
95.86155507093572,38.22527805795094,0
75.01365838958247,30.60326323428011,0
82.30705337399482,76.48196330235604,1
69.36458875970939,97.71869196188608,1
39.53833914367223,76.03681085115882,0
53.9710521485623,89.20735013750205,1
69.07014406283025,52.74046973016765,1
67.94685547711617,46.67857410673128,0
```

图 3.9 数据存放

每个数据点是一行，每一行中前面两个数据表示 x 坐标和 y 坐标，最后一个数据表示其类别。

我们先从 data.txt 文件中读取数据，使用非常简单的 python 读取 txt 的方法就能够实现。

```
with open('data.txt', 'r') as f:
    data_list = f.readlines()
    data_list = [i.split('\n')[0] for i in data_list]
    data_list = [i.split(',') for i in data_list]
    data = [(float(i[0]), float(i[1]), float(i[2])) for i in data_list]
```

然后通过 matplotlib 能够简单地将数据画出来。

```
x0 = list(filter(lambda x: x[-1] == 0.0, data))
x1 = list(filter(lambda x: x[-1] == 1.0, data))
plot_x0_0 = [i[0] for i in x0]
plot_x0_1 = [i[1] for i in x0]
plot_x1_0 = [i[0] for i in x1]
plot_x1_1 = [i[1] for i in x1]

plt.plot(plot_x0_0, plot_x0_1, 'ro', label='x_0')
plt.plot(plot_x1_0, plot_x1_1, 'bo', label='x_1')
plt.legend(loc='best')
```

首先将两个类别分开，然后将所有的数据点画出就能够得到图3.10。

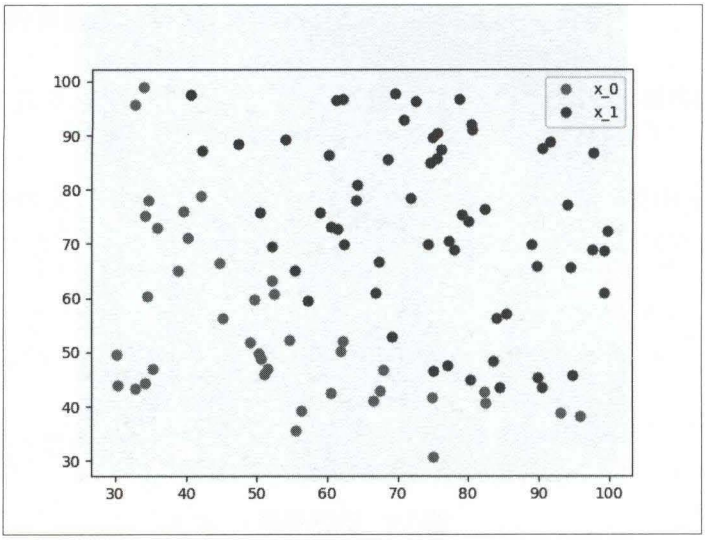


图 3.10　数据点

从图3.10中我们可以明显看出这些数据点被分为两个类：一类用红色的点，一类用蓝色的点，我们希望通过 Logistic 回归将它们分开。

接下来定义 Logistic 回归的模型，以及二分类问题的损失函数和优化方法。

```
class LogisticRegression(nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.lr = nn.Linear(2, 1)
        self.sm = nn.Sigmoid()

    def forward(self, x):
        x = self.lr(x)
        x = self.sm(x)
        return x

logistic_model = LogisticRegression()
if torch.cuda.is_available():
    logistic_model.cuda()

criterion = nn.BCELoss()
optimizer = torch.optim.SGD(logistic_model.parameters(), lr=1e-3,
                            momentum=0.9)
```

这里 nn.BCELoss 是二分类的损失函数，torch.optim.SGD 是随机梯度下降优化函数。

然后训练模型，并且间隔一定的迭代次数输出结果。

```
for epoch in range(50000):
    if torch.cuda.is_available():
        x = Variable(x_data).cuda()
        y = Variable(y_data).cuda()
    else:
        x = Variable(x_data)
        y = Variable(y_data)
    # ==================forward==================
    out = logistic_model(x)
    loss = criterion(out, y)
    print_loss = loss.data[0]
    mask = out.ge(0.5).float()
    correct = (mask == y).sum()
    acc = correct.data[0] / x.size(0)
    # ==================backward=================
```

```
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch+1) % 1000 == 0:
        print('*'*10)
        print('epoch {}'.format(epoch+1))
        print('loss is {:.4f}'.format(print_loss))
        print('acc is {:.4f}'.format(acc))
```

其中 `mask=out.ge(0.5).float()` 是判断输出结果如果大于 0.5 就等于 1，小于 0.5 就等于 0，通过这个来计算模型分类的准确率。

训练完成我们可以得到图 3.11所示的 loss 和准确率。因为数据相对简单，同时我们使用的是也是简单的线性 Logistic 回归，loss 已经降得相对较低，同时也有 91% 的准确率。

```
epoch 48000
loss is 0.2549
acc is 0.9100
**********
epoch 49000
loss is 0.2538
acc is 0.9100
**********
epoch 50000
loss is 0.2527
acc is 0.9100
```

图 3.11　loss 和准确率

我们可以将这条直线画出来，因为模型中学习的参数 w_1，w_2 和 b 其实构成了一条直线 $w_1x + w_2y + b = 0$，在直线上方是一类，在直线下方又是一类。我们可以通过下面的方式将模型的参数取出来，并将直线画出来，如图 3.12所示。

```
w0, w1 = logistic_model.lr.weight[0]
w0 = w0.data[0]
w1 = w1.data[0]
b = logistic_model.lr.bias.data[0]
plot_x = np.arange(30, 100, 0.1)
```

```
plot_y = (-w0 * plot_x - b) / w1
plt.plot(plot_x, plot_y)
plt.show()
```

通过图 3.12 我们可以看出这条直线基本上将这两类数据都分开了。

以上我们介绍了分类问题中的二分类问题和 Logistic 回归算法，一般来说，Logistic 回归也可以处理多分类问题，但最常见的还是应用在处理二分类问题上，下面我们将介绍一下使用神经网络算法来处理多分类问题。

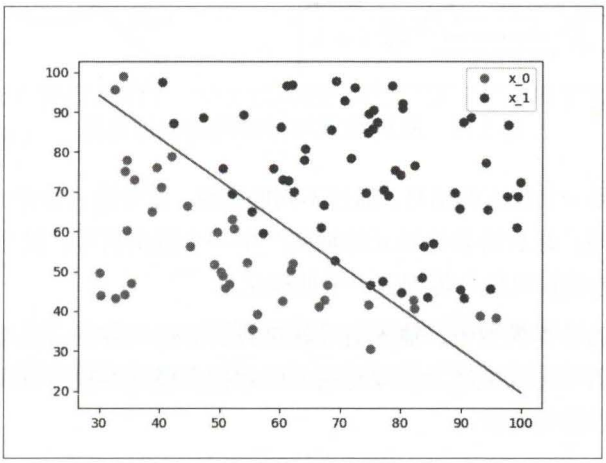


图 3.12 直线分开两类数据点

3.4 简单的多层全连接前向网络

前面我们介绍了机器学习领域中两个最基本的算法：一个是线性回归，一个是 Logistic 回归，下面的部分将聚焦于机器学习中的深度学习。我们知道深度学习的前身就是神经网络，但是由于之前的计算能力与数据量不足、传统机器学习方法如支持向量机完美的数学解释等原因，使得深度学习一直没有发展起来，直到最近几年才有了爆发式的发展，首先从它们之间的过渡——神经网络算法入手。

3.4.1 模拟神经元

神经网络最开始是受到了模拟脑神经元的启发，但是现在已经发展成了机器学习中的一个重要算法，我们先简要地介绍脑神经科学，接着介绍神经网络。

脑中的一个计算单元是一个简单的脑神经元，在人脑的神经系统中，大约有 8.6×10^{10} 个神经元，它们被 $10^{14} \sim 10^{15}$ 个突触相连，图 3.13能够简要地展示一下脑神经元与神经网络的相似之处。

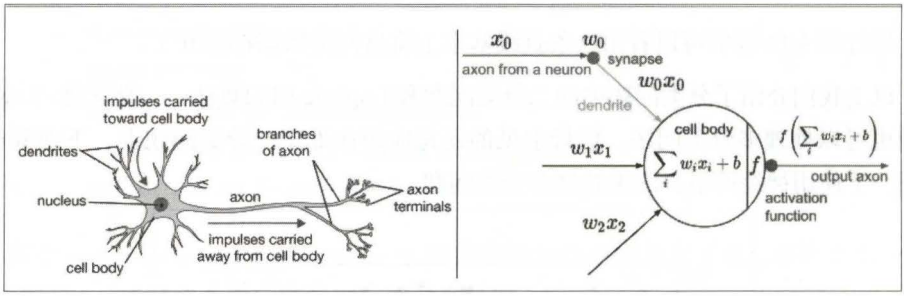


图 3.13　脑神经元与神经网络相似之处

脑神经元收到一个输入的信号，通过不同的突触，信号进入神经元，接着通过神经元内部的激活处理，最后沿着神经元的轴突产生一个输出信号，这个轴突通过下一个神经元的突触相连将输出信号传到下一个神经元。

在神经网络的计算模型中，输入信号就是我们的输入数据，模型的参数就相当于突触，然后输入信号传入神经元就像是输入的数据和模型参数进行线性组合，然后经过激活函数，最后传出模型。

这里的模型参数通过学习可以控制输入数据传入神经元的强度，激活函数就是神经元内部的激活处理，最后将结果输出变成第二层网络的输入。我们之前讲过的 Sigmoid 函数，也就是 logistic 函数 $\sigma(x) = \frac{1}{1+e^{-x}}$ 就是一种激活函数，它能够将输入的数据转化到 $0 \sim 1$ 之间。

但以上只是一个粗糙的脑神经元简述，因为在大脑里面，有很多不同类型的神经元，每种神经元的性质都不一样，同时神经突触还是一个复杂的动态系统，而且神经元的输出也是有时效性的，总之这只是一个简化的神经系统，但是可以给我们一些关于神经网络的启发。

3.4.2　单层神经网络的分类器

一个简单的一层神经网络的数学形式是特别简单的，可以看成一个线性运算再加上一个激活函数，正如前面讲过的，一个神经元可以对一个输入进行不同的操作，可以是“喜欢”（激活变大）或者是“不喜欢”（激活变小），正是由于激活函数的作用，所以我们可以将一层神经网络用做分类器，正样本就让激活函数激活变大，负样本就让激活函数激活变小。

例如我们可以使用 Sigmoid 激活函数做一个二分类问题，这个时候使用数学形式可以得到 $\sigma(\sum_{i=1}^{n} w_i x_i + b)$ 表示 $P(y_i = 1|x_i, w)$，即输出为 1 类的概率，那么输出为 0 类的概率为 $1 - P(y_i = 1|x_i, w)$，是不是看着很熟悉呢？将 $P(y_i = 1|x_i, w)$ 展开，即

$$P(y_i = 1|x_i, w) = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{3.30}$$

$$= \frac{1}{1 + e^{-(w_i x_i + b)}} \tag{3.31}$$

$$= \frac{e^{w_i x_i + b}}{1 + e^{w_i x_i + b}} \tag{3.32}$$

这就是前面讲过的 Logistic 回归，所以 Logistic 回归只是一个使用了 Sigmoid 作为激活函数的一层神经网络。

3.4.3 激活函数

上面我们知道了使用 Sigmoid 激活函数的一层神经网络就是 Logistic 回归，可以看到激活函数也占据着重要的地位，下面我们将介绍一下神经网络中各种常用的激活函数。

1.Sigmoid

Sigmoid 非线性激活函数的数学表达式是 $\sigma(x) = \frac{1}{1+e^{-x}}$，其图形如图 3.14所示。目前我们知道 Sigmoid 激活函数是将一个实数输入转化到 $0 \sim 1$ 之间的输出，具体来说也就是将越大的负数转化到越靠近 0，越大的正数转化到越靠近 1。历史上 Sigmoid 函数频繁地使用，因为其具有良好的解释性。

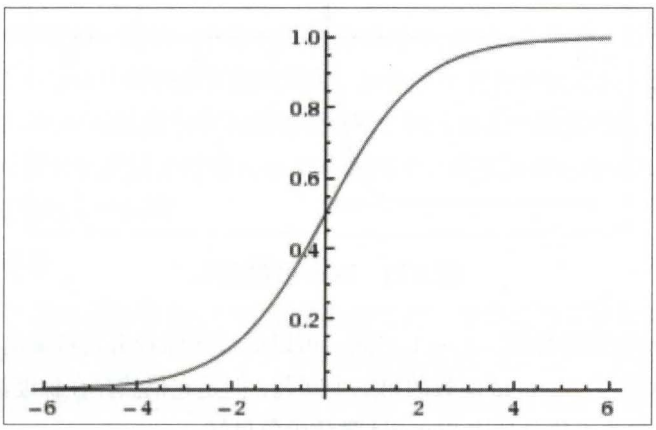


图 3.14 Sigmoid 函数图形

但是最近几年，Sigmoid 激活函数已经越来越少地被人使用了，主要是因为 Sigmoid 函数有以下两大缺点。

（1）Sigmoid 函数会造成梯度消失。一个非常不好的特点就是 Sigmoid 函数在靠近 1 和 0 的两端时，梯度会几乎变成 0，我们前面讲过梯度下降法通过梯度乘上学习率来更新参数，因此如果梯度接近 0，那么没有任何信息来更新参数，这样就会造成模型不收敛。另外，如果使用 Sigmoid 函数，那么需要在初始化权重的时候也必须非常小心。如果初始化的时候权重太大，那么经过激活函数也会导致大多数神经元变得饱和，没有办法更新参数。

（2）Sigmoid 输出不是以 0 为均值，这就会导致经过 Sigmoid 激活函数之后的输出，作为后面一层网络的输入的时候是非 0 均值的，这个时候如果输入进入下一层神经元的时候全是正的，这就会导致梯度全是正的，那么在更新参数的时候永远都是正梯度。怎么理解呢？比如进入下一层神经元的输入是 x，参数是 w 和 b，那么输出就是 $f = wx + b$，这个时候 $\nabla f(w) = x$，所以如果 x 是 0 均值的数据，那么梯度就会有正有负。但是这个问题并不是太严重，因为一般神经网络在训练的时候都是按 batch（批）进行训练的，这个时候可以在一定程度上缓解这个问题，所以说虽然 0 均值这个问题会产生一些不好的影响，但是总体来讲跟上一个缺点：梯度消失相比还是要好很多。

2.Tanh

Tanh 激活函数是上面 Sigmoid 激活函数的变形，其数学表达为 $tanh(x) = 2\sigma(2x) - 1$，图形如图3.15所示。

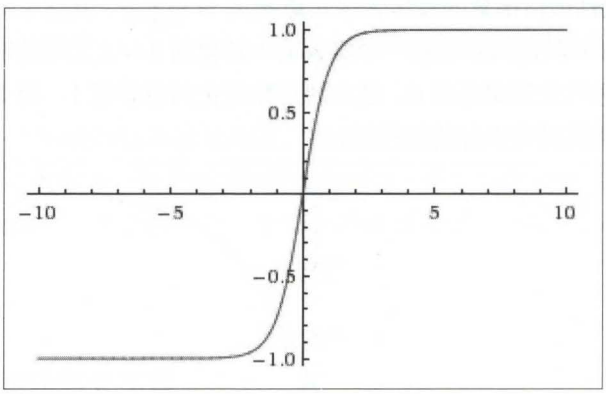


图 3.15　Tanh 函数图形

它将输入的数据转化到 $-1 \sim 1$ 之间，可以通过图像看出它将输出变成了 0 均值，在一定程度上解决了 Sigmoid 函数的第二个问题，但是它仍然存在梯度消失的问题。因此实际上 Tanh 激活函数总是比 Sigmoid 激活函数更好。

3.ReLU

ReLU 激活函数（Rectified Linear Unit）近几年变得越来越流行，它的数学表达式为 $f(x) = max(0, x)$，换句话说，这个激活函数只是简单地将大于 0 的部分保留，将小于 0 的部分变成 0，它的图形如图 3.16所示。

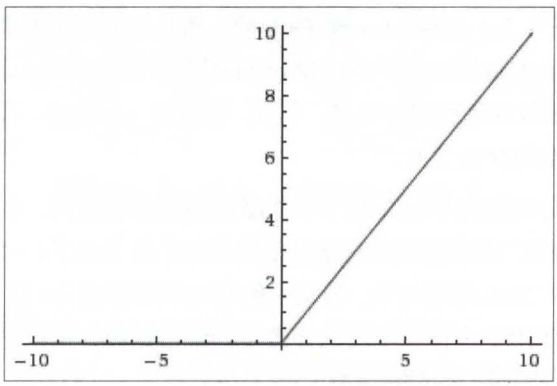


图 3.16 ReLU 函数图形

下面我们简单来介绍一下 ReLU 激活函数的优缺点。

ReLU 的优点：

（1）相比于 Sigmoid 激活函数和 Tanh 激活函数，ReLU 激活函数能够极大地加速随机梯度下降法的收敛速度，这因为它是线性的，且不存在梯度消失的问题。

（2）相比于 Sigmoid 激活函数和 Tanh 激活函数的复杂计算而言，ReLU 的计算方法更加简单，只需要一个阈值过滤就可以得到结果，不需要进行一大堆复杂的运算。

ReLU 的缺点：

训练的时候很脆弱，比如一个很大的梯度经过 ReLU 激活函数，更新参数之后，会使得这个神经元不会对任何数据有激活现象。如果发生这种情况之后，经过 ReLU 的梯度永远都会是 0，也就意味着参数无法再更新了，因为 ReLU 激活函数本质上是一个不可逆的过程，因为它会直接去掉输入小于 0 的部分。在实际操作中可以通过设置比较小的学习率来避免这个小问题。

4.Leaky ReLU

Leaky ReLU 激活函数是 ReLU 激活函数的变式，主要是为了修复 ReLU 激活函数中训练比较脆弱的这个缺点，不将 $x < 0$ 的部分变成 0，而给它一个很小的负的斜率，比如 0.01，它的数学形式可以表现为 $f(x) = I(x < 0)(\alpha x) + I(x \geq 0)(x)$，其中 α 是一个很小的常数，这样就可以使得输入小于 0 的时候也有一个小的梯度。关于 Leaky ReLU 激活函数的效果，众说纷纭，一些实验证明很好，一些实验证明并不好。

同时也有人提出可以对 α 进行参数化处理，也就是说可以在网络的训练过程中对 α 也进行更新，但是否对所有情况都有效，目前也不清楚。

5.Maxout

另外一种激活函数的类型并不是 $f(wx+b)$ 作用在一种输出结果的形式，而是 $max(w_1x+b_1, w_2x+b_2)$ 这种 Maxout 的类型，可以发现 ReLU 激活函数只是 Maxout 中 $w_1=0, b_1=0$ 的特殊形式。因此 Maxout 既有着 ReLU 激活函数的优点，同时也避免了 ReLU 激活函数训练脆弱的缺点。不过，它也有一个缺点，那就是它加倍了模型的参数，导致了模型的存储变大。

通过上面的部分我们简单地介绍了一些激活函数的优缺点，在实际我们使用较多的还是 ReLU 激活函数，但是需要注意学习率的设定不要太大了；一定不要使用 Sigmoid 激活函数，可以试试 Tanh 激活函数，但是一般它的效果都比 ReLU 和 Maxout 差。最后一点，我们在实际使用中也很少使用混合类型的激活函数，也就是说一般在同一个网络中我们都使用同一种类型的激活函数。

3.4.4 神经网络的结构

神经网络是一个由神经元构成的无环图，换句话说一些神经元的输出会变成另外一些神经元的输入，环是不被允许的，因为这样会造成网络中的无限循环。同时神经网络一般是以层来组织的，最常见的神经网络就是全连接神经网络，其中两个相邻层中每一个层的所有神经元和另外一个层的所有神经元相连，每个层内部的神经元不相连，如图 3.17和图 3.18所示。

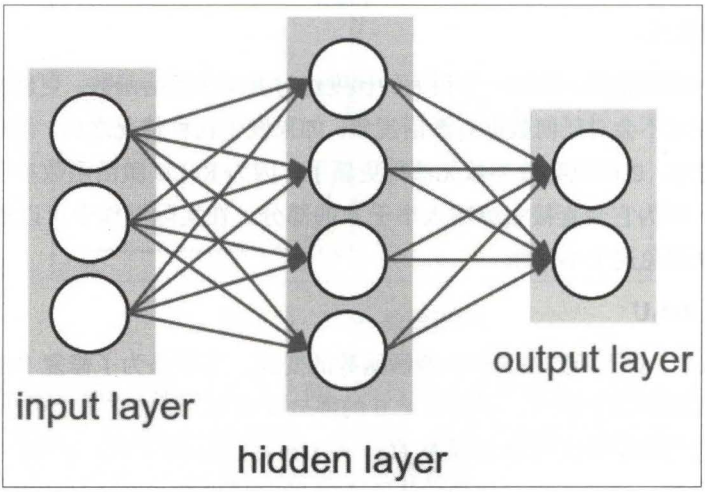


图 3.17　hidden layer

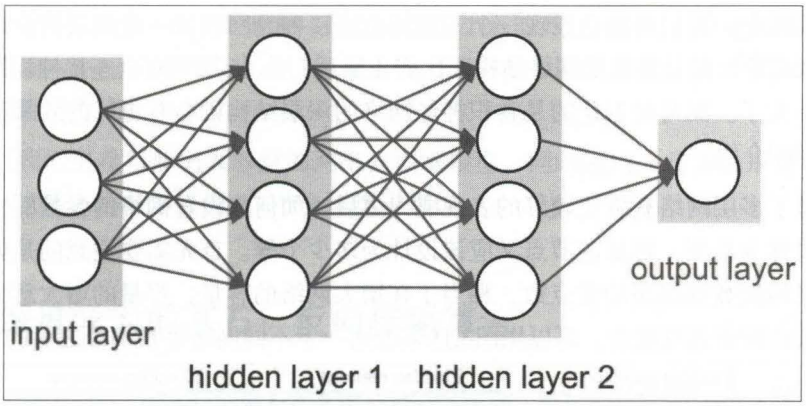


图 3.18 hidden layer 1 与 hidden layer 2

在之前的线性模型和 Logistic 回归中，我们已经接触到了 `nn.Linear(in, out)`，它就是在 PyTorch 里用来表示一个全连接神经网络层的函数，比如输入层 4 个节点，输出 2 个节点，可以用 `nn.Linear(4, 2)` 来表示，同时 `nn.Linear(in, out, bias=False)` 可以不使用偏置，默认是 True。

一般而言，N 层神经网络并不会把输入层算进去，因此一个一层的神经网络是指没有隐藏层、只有输入层和输出层的神经网络。Logistic 回归就是一个一层的神经网络。

输出层一般是没有激活函数的，因为输出层通常表示一个类别的得分或者回归的一个实值的目标，所以输出层可以是任意的实数。

3.4.5 模型的表示能力与容量

前面我们通过脑神经结构引出了神经网络的层结构，如果从数学的角度来解释神经网络，那么神经网络就是由网络中的参数决定的函数簇。所谓的函数簇就是一系列的函数，这些函数都是由网络的参数决定的。提出了函数簇之后，我们就想明确这个函数簇的表达能力，也就是我们想知道是否有函数是这个函数簇无法表达的？

这个问题在 1989 年就被人证明过，拥有至少一个隐藏层的神经网络可以逼近任何的连续函数。如果一个只有一层隐藏层的神经网络就能够逼近任何连续函数，为什么我们还要使用更多层的网络呢？

这个问题可以这么去理解，理论上讲增加的网络层可以看成是一系列恒等变换的网络层，也就是说这些网络层对输入不做任何变换，这样这个深层的网络结构至少能够达到与这个浅层网络相同的效果；同时在实际使用中我们也发现更深层的网络具有更好的表现力，同时有着更好的优化结果。

在实际中，我们可能会发现一个三层的全连接神经网络比一个两层的全连接神经网络表现更好，但是更深的网络结构，比如 4 层、5 层、6 层等对全连接神经网络效果提升就不大了。而与此对比的是卷积神经网络的深层结构则会有更好的效果，下一章我们会详细介绍。

知道了多层网络有着比较好的表现能力之后，如何来设置网络的参数呢？比如我们应该设计为几层，每层的节点又应该设计为多少个等。首先需要注意的是如果我们增大网络的层数和每层的节点数，相当于在增大网络的容量，容量的增大意味着网络有着更大的潜在表现能力，可以用图3.19来说明一下不同网络容量训练之后的效果。

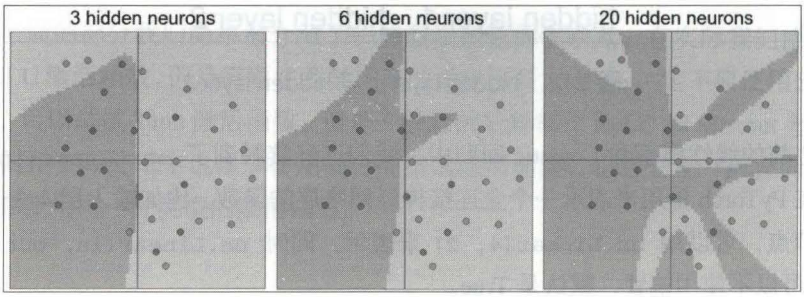


图 3.19　不同网络容量训练结果

上面三张图分别是三个网络模型做二分类得到的结果，每个网络模型都是一个隐藏层，但是每个隐藏层的节点数目不一样，从左到右分别是 3 个、6 个和 20 个隐藏节点，这三个模型训练之后得到的结果完全不一样，可以看到隐藏节点越多的模型能够表示更加复杂的模型，然而根据我们想要的结果，其实最左边的模型才是最好的，最右边的模型虽然有着更加复杂的形状，但是它忽略了潜在的数据关系，将噪声的干扰放大了，这种效果被称为过拟合 (overfitting)。

从最右边的图片我们可以看出虽然模型成功地将红色的点和绿色的点完全分开，但是却付出了很大的代价，将区域分割成了很多分离的区域，最左边的图虽然模型没能将所有的点都完全分开，但是形成了两个大区域，在实际应用中，这样的结果抗噪能力和泛化能力反而更强。

通过上面的讨论，我们知道了如果数据不太复杂，那么容量较小的模型反而有着更好的效果，是不是我们可以用小的模型去处理呢？答案并非如此，首先通过这样的方式没有办法很好地衡量到底多小的模型才算是小的模型，其次小的模型在使用如梯度下降法等训练的时候通常更难。

因为神经网络的损失函数一般是非凸的，容量小的网络更容易陷入局部极小点而达不到最优的效果，同时这些局部最小点的方差特别大，换句话说，也就是每个局部最优点的差异都特别大，所以你在训练网络的时候训练 10 次可能得到的结果有很大的差

异。但是对于容量更大的神经网络，它的局部极小点的方差特别小，也就是说训练多次虽然可能陷入不同的局部极小点，但是它们之间的差异是很小的，这样训练就不会完全依靠随机初始化。

所以我们更希望使用大容量的网络去训练模型，同时运用一些方法来控制网络的过拟合，这些方法在之后的 3.7 节会详细讲解。

3.5 深度学习的基石：反向传播算法

前一章我们介绍了前向传播的多层全连接神经网络，一个核心的问题就是给出了损失函数 f，我们需要更新参数那就需要算出 f 的梯度 $\nabla f(x)$，那么我们如何能够有效地求出这个梯度呢？

反向传播算法就是一个有效地求解梯度的算法，本质上其实就是一个链式求导法则的应用，然而这个如此简单而且显而易见的方法却是在 Roseblatt 提出感知器算法后将近 30 年才被发明和普及的，对此 Bengio 这样说道："很多看似显而易见的想法只有在事后才变得显而易见。"

下面我们就来详细将一讲什么是反向传播算法。

3.5.1 链式法则

首先来简单地介绍一下链式法则，考虑一个简单的函数，比如 $f(x,y,z)=(x+y)z$，我们当然可以直接求出这个函数的微分，但是这里我们要使用链式法则，令 $q=x+y$，那么 $f=qz$，对于这两个式子，分别求出它们的微分，$\frac{\partial f}{\partial q}=z, \frac{\partial f}{\partial z}=q$，同时 q 是 x 和 y 的求和，所以能够得到 $\frac{\partial q}{x}=1, \frac{\partial q}{y}=1$。我们关心的问题是 $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$，链式法则告诉我们如何来计算出它们的值。

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial x} \tag{3.33}$$

$$\frac{\partial f}{\partial y}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial y} \tag{3.34}$$

$$\frac{\partial f}{\partial z}=q \tag{3.35}$$

通过链式法则知道，如果需要对其中的元素求导，那么可以一层一层求导，然后将结果乘起来，这就是链式法则的核心，也是反向传播算法的核心。

3.5.2 反向传播算法

了解了链式法则，我们就可以开始介绍反向传播算法，本质上反向传播算法只是链式法则的一个应用。还是使用之前那个相同的例子 $q = x + y, f = qz$，通过计算图可以将这个计算过程表达出来，如图 3.20所示。

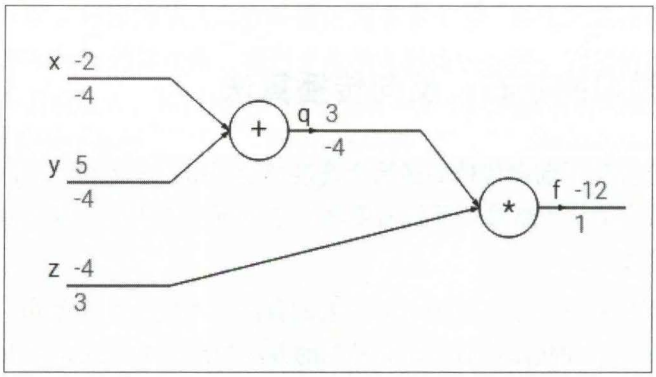


图 3.20 计算图

上面的数字表示其数值，下面的数字表示求出的梯度，我们可以一步一步地看看反向传播算法的实现。首先从最后开始，梯度当然是 1，然后计算 $\frac{\partial f}{\partial q} = z = -4, \frac{\partial f}{\partial z} = q = 3$，接着计算 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 \times 1 = -4, \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 \times 1 = -4$，这样一步一步地就求出了 $\nabla f(x, y, z)$。

直观上看反向传播算法是一个优雅的局部过程，每次求导只是对当前的运算求导，求解每层网络的参数都是通过链式法则将前面的结果求出不断迭代到这一层的，所以说这是一个传播过程。

3.5.3 Sigmoid 函数举例

下面我们通过 Sigmoid 函数来演示反向传播过程在一个复杂的函数上是如何进行的。

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \tag{3.36}$$

我们需要求解出 $\frac{\partial f}{\partial w_0}, \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}$。首先将这个函数抽象成一个计算图来表示，即

$$f(x) = \frac{1}{x} \tag{3.37}$$

$$f_c(x) = c + x \tag{3.38}$$

$$f(x) = e^x \tag{3.39}$$

$$f_a(x) = ax \tag{3.40}$$

这样就能够画出如图 3.21所示的计算图。

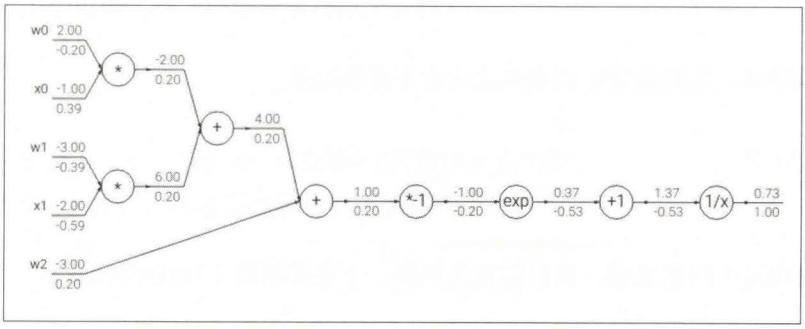


图 3.21 计算图

同样图中的数字表示数值，下面的数字表示梯度，从后往前计算一下各个参数的梯度。首先最后面的梯度是 1,，然后经过 $\frac{1}{x}$ 这个函数，这个函数的梯度是 $-\frac{1}{x^2}$，所以往前传播的梯度是 $1 \times -\frac{1}{1.37^2} = -0.53$，然后 $+1$ 这个操作，梯度不变，接着是 e^x 这个运算，它的梯度就是 $-0.53 \times e^{-1} = -0.2$，这样不断往后传播就能够求得每个参数的梯度。

通过链式法则我们引入了反向传播算法，从上面的例子详细地运算了一下反向传播算法，这是深度学习中优化算法的核心，因为所有基于梯度的优化算法都需要计算每个参数的梯度，下面详细讲一下各种优化算法的变式。

3.6 各种优化算法的变式

在前面讲 `torch.optim` 的部分，我们简单介绍了梯度下降法，下面会先讲一讲梯度下降法的原理，再详细地介绍各种基于梯度的优化方法。

3.6.1 梯度下降法

首先，我们知道梯度下降法的更新公式如下：

$$\theta^i = \theta^{i-1} - \eta \nabla L(\theta^{i-1}) \tag{3.41}$$

现在希望通过实际理论来证明这样更新参数能够达到最优的效果。

重新表述一下问题，希望求解下面这个方程：

$$\theta^* = \arg\min_{\theta} L(\theta) \tag{3.42}$$

也就是说，我们希望更新参数之后有下面的结果：

$$L(\theta_0) > L(\theta_1) > L(\theta_2) > \cdots \tag{3.43}$$

在解决这个问题之前，我们需要先回顾一下泰勒级数（Taylor Series）。

对于任何一个无限可微函数 $h(x)$，在一个点 $x = x_0$ 附近，有以下的泰勒级数：

$$h(x) = \sum_{k=0}^{\infty} \frac{h^{(k)}(x_0)}{k!}(x - x_0)^k \tag{3.44}$$

$$= h(x_0) + h^{'}(x_0)(x - x_0) + \frac{h^{''}(x_0)}{2!}(x - x_0)^2 + \cdots \tag{3.45}$$

当 x 足够接近 x_0，有下面的近似：

$$h(x) \approx h(x_0) + h^{'}(x_0)(x - x_0) \tag{3.46}$$

对于多元泰勒级数，有以下的公式：

$$h(x, y) = h(x_0, y_0) + \frac{\partial h(x_0, y_0)}{\partial x}(x - x_0) + \frac{\partial h(x_0, y_0)}{\partial y}(y - y_0) + \cdots \tag{3.47}$$

同样当 x 和 y 足够接近 x_0 和 y_0 的时候，有以下的近似：

$$h(x, y) \approx h(x_0, y_0) + \frac{\partial h(x_0, y_0)}{\partial x}(x - x_0) + \frac{\partial h(x_0, y_0)}{\partial y}(y - y_0) \tag{3.48}$$

现在假设参数 θ 有两个变量 θ_1, θ_2，那么由上面的二元泰勒级数知道，对于一个点 (a, b)，对于一个足够小的范围，有以下近似：

$$L(\theta) = L(a, b) + \frac{\partial L(a, b)}{\partial \theta_1}(\theta_1 - a) + \frac{\partial L(a, b)}{\partial \theta_2}(\theta_2 - b) \tag{3.49}$$

令 $s = L(a,b), u = \frac{\partial L(a,b)}{\partial \theta_1}, v = \frac{\partial L(a,b)}{\partial \theta_2}$，那么 $L(\theta)$ 有以下的简单表达：

$$L(\theta) \approx s + u(\theta_1 - a) + v(\theta_2 - b) \tag{3.50}$$

我们知道 s，u 和 v 都是常数，所以希望找到 θ_1, θ_2 使得 $L(\theta)$ 最小，同时 θ_1, θ_2 都是在 (a, b) 的一个小范围之内，所以要满足以下条件：

$$(\theta_1 - a)^2 + (\theta_2 - b)^2 \leq \epsilon^2 \tag{3.51}$$

再换元，将 $(\theta_1 - a) = \Delta\theta_1, (\theta_2 - b) = \Delta\theta_2$，同时由于 s 是一个与 θ_1, θ_2 没有关系的常量，所以可以去掉它，这样就能够得到下面的式子：

$$L(\theta) \Leftrightarrow u\Delta\theta_1 + v\Delta\theta_2 \tag{3.52}$$

$$\Delta\theta^2 + \Delta\theta^2 \leq \epsilon^2 \tag{3.53}$$

对于上面这个式子，可以将 (u, v) 看做一个向量，同时 $(\Delta\theta_1, \Delta\theta_2)$ 也是一个向量，所以 $(u, v) \cdot (\Delta\theta_1, \Delta\theta_2) = u\Delta\theta_1 + v\Delta\theta_2$。那么怎么能够求到这个内积的最小值呢？很简单，只需要保证 $(\Delta\theta_1, \Delta\theta_2)$ 是 (u, v) 的反方向，也就是 $(\Delta\theta_1, \Delta\theta_2) = -(u, v)$，但是由于需要把 $\Delta\theta_1, \Delta\theta_2$ 都限制在一个小的范围内，所以要对其范数做一个限制，也就是：

$$(\Delta\theta_1, \Delta\theta_2) = -\eta(u, v) \tag{3.54}$$

$$\Longleftrightarrow (\theta_1, \theta_2) = (a, b) - \eta(u, v) \tag{3.55}$$

所以只要在一个足够小的范围内，更新参数之后就能取得一个更小的值，更新公式如下所示：

$$(\theta_1, \theta_2) = (a, b) - \eta(\frac{\partial L(a,b)}{\partial \theta_1}, \frac{\partial L(a,b)}{\partial \theta_2}) \tag{3.56}$$

所以只要取一个特别小的学习率 η，就能够保证 θ_1, θ_2 在一个足够靠近 (a, b) 的范围内，这就实现了梯度下降法。

下面介绍一些常见的梯度下降法的变式。

3.6.2 梯度下降法的变式

1.SGD

随机梯度下降法是梯度下降法的一个小变形，就是每次使用一批 (batch) 数据进行梯度的计算，而不是计算全部数据的梯度，因为现在深度学习的数据量都特别大，所以每次都计算所有数据的梯度是不现实的，这样会导致运算时间特别长，同时每次都计算全部的梯度还失去了一些随机性，容易陷入局部误差，所以使用随机梯度下降法可能每次都不是朝着真正最小的方向，但是这样反而容易跳出局部极小点。

2.Momentum

第二种优化方法就是在随机梯度下降的同时，增加动量 (Momentum)。这来自于物理中的概念，可以想象损失函数是一个山谷，一个球从山谷滑下来，在一个平坦的地势，球的滑动速度就会慢下来，可能陷入一些鞍点或者局部极小值点，如图 3.22所示。

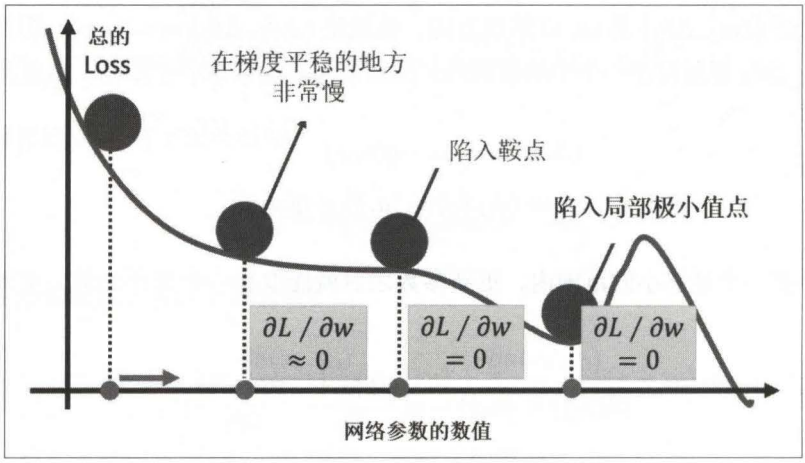


图 3.22 增加动量

这个时候给它增加动量就可以让它从高处滑落时的势能转换为平地的动能，相当于惯性增加了小球在平地滑动的速度，从而帮助其跳出鞍点或者局部极小点。

动量怎么计算呢？动量的计算基于前面梯度，也就是说参数更新不仅仅基于当前的梯度，也基于之前的梯度，可以用图3.23来简单地说明。

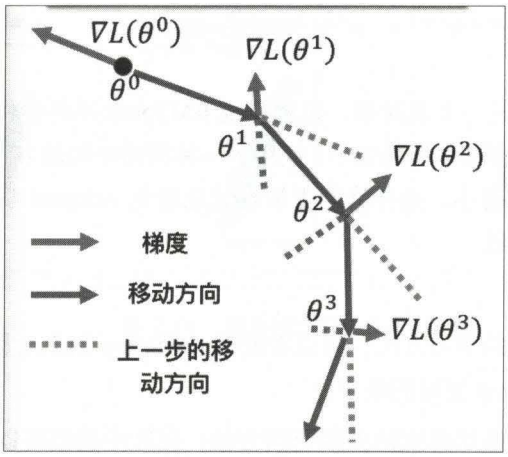


图 3.23 动量的计算

记住我们更新的是梯度的负方向，红色表示梯度，蓝色表示更新方向，图中绿色的虚线就是动量，可以看到也就是之前一次梯度的负方向。

除此之外，对于动量还有一个变形，即 Nesterov。我们在更新参数的时候需要计算梯度，传统的动量方法是计算当前位置的梯度，但是 Nesterov 的方法是计算经过动量更新之后的位置的梯度。

3.Adagrad

这是一种自适应学习率 (adaptive) 的方法，它的公式是：

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2 + \epsilon}} g^t \tag{3.57}$$

通过式（3.57），我们可以看到学习率在不断变小，且受每次计算出来的梯度影响，对于梯度比较大的参数，它的学习率就会变得相对更小，里面的根号特别重要，没有这个根号算法表现非常差。同时 ϵ 是一个平滑参数，通常设置为 $10^{-4} \sim 10^{-8}$，这是为了避免分母为 0。

自适应学习率的缺点就是在某些情况下一直递减的学习率并不好，这样会造成学习过早停止。

4.RMSprop

这是一种非常有效的自适应学习率的改进方法，它的公式是：

$$cache^t = \alpha * cache^{t-1} + (1-\alpha)(g^t)^2 \tag{3.58}$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{cache^t + \epsilon}} g^t \tag{3.59}$$

这里多了一个 α，这是一个衰减率，也就是说 RMSprop 不再会将前面所有的梯度平方求和，而是通过一个衰减率将其变小，使用了一种滑动平均的方式，越靠前面的梯度对自适应的学习率影响越小，这样就能更加有效地避免 Adagrad 学习率一直递减太多的问题，能够更快地收敛。

5.Adam

这是一种综合型的学习方法，可以看成是 RMSprop 加上动量 (Momentum) 的学习方法，达到比 RMSProp 更好的效果。

以上介绍了多种基于梯度的参数更新方法，实际中我们可以使用 Adam 作为默认的优化算法，往往能够达到比较好的效果，同时 SGD+Momentum 的方法也值得尝试。

3.7 处理数据和训练模型的技巧

在开始训练网络之前，良好的数据预处理和参数初始化能够很快达到事半功倍的结果。在模型训练中采用一些训练技巧，能够使得模型最后达到 state-of-art 的效果，这一节我们讲一讲处理数据和训练模型的技巧。

3.7.1 数据预处理

1. 中心化

数据预处理中一个最常见的处理办法就是每个特征维度都减去相应的均值实现中心化，这样可以使得数据变成 0 均值，特别对于一些图像数据，为了方便我们将所有的数据都减去一个相同的值。

2. 标准化

在使得数据都变成 0 均值之后，还需要使用标准化的做法让数据不同的特征维度都有着相同的规模。有两种常用的方法：一种是除以标准差，这样可以使得新数据的分布接近标准高斯分布；还有一种做法就是让每个特征维度的最大值和最小值按比例缩放到 $-1 \sim 1$ 之间。

如果知道输入不同特征有着不同的规模，那就需要使用标准化的做法让它们处于同一个规模下面，这对于机器学习算法而言是非常重要的。

我们可以通过图3.24看看每种做法处理完数据之后的效果。

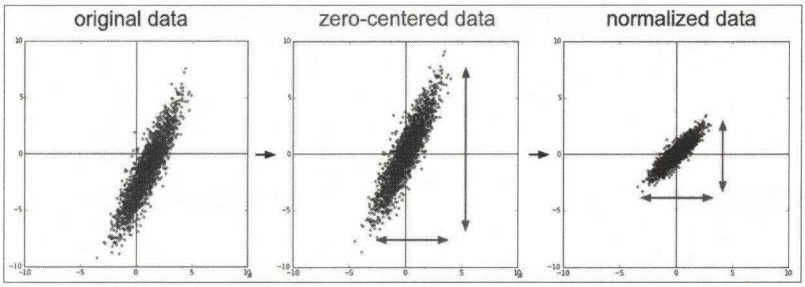


图 3.24　数据处理效果

3.PCA

PCA 是另外一种处理数据的方法，在进行这一步以前，首先会将数据中心化，然后计算数据的协方差矩阵，这一步特别简单，假设输入是 $X = N \times D$，那么通过 $\frac{X^T X}{N}$ 能够得到这个协方差矩阵，可以验证一下这个结果的正确性，而且这个协方差矩阵是对称半正定的，可以通过这个协方差矩阵来进行奇异值分解（SVD），然后对数据进行去相关性，将其投影到一个特征空间，我们能够取一些较大的、主要的特征向量来降低数据的维数，去掉一些没有方差的维度，这也叫做主成分分析（PCA）。

这个操作对于一些线性模型和神经网络，都能取得良好的效果。

4. 白噪声

白噪声也是一种处理数据的方式，首先会跟 PCA 一样将数据投影到一个特征空间，然后每个维度除以特征值来标准化这些数据，直观上就是一个多元高斯分布转化到了一个 0 均值，协方差矩阵为 1 的多元高斯分布。

具体的 PCA 和白噪声处理之后的效果如图3.25所示。

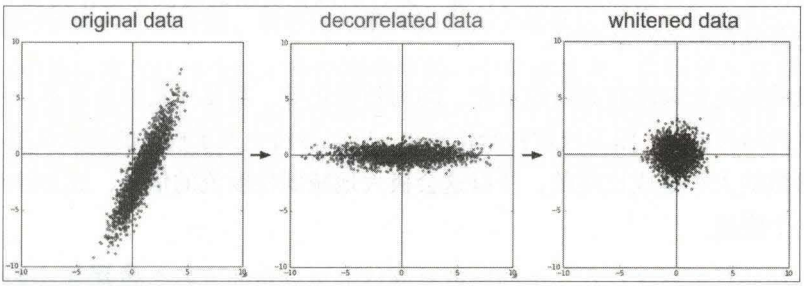


图 3.25　PCA 和白噪声处理效果

图 3.25 形象地展示了 PCA 和白噪声处理之后的效果，但是白噪声的处理会增强数据中的噪声，因为其增强了数据中的所有维度，包括了一些方差很小的不相关的维度。

在实际处理数据中，中心化和标准化都特别重要。我们计算训练集的统计量比如均值，然后将这些统计量应用到测试集和验证集当中。但是 PCA 和白噪声在卷积网络

中基本不使用，因为卷积网络可以自动学习如何提取这些特征而不需要人工再去对其进行干预。

3.7.2 权重初始化

前面介绍了数据的预处理，在进入网络训练之前，需要做参数的预处理。下面介绍预处理的策略。

1. 全 0 初始化

首先从一个最直观，但是不应该采用的策略入手，那就是将参数全部初始化为 0。

解释一下为什么不能采用这一种策略。首先，我们并不知道训练之后的网络最后权重更新的是多少，但是知道数据在进入网络之前经过了合适的预处理，所以我们可以假设最后的权重有一半是正的，一半是负的，所以将参数全部初始化为 0 似乎是一个非常好的选择。但这是不对的，因为如果神经网络中每个权重都被初始化成相同的值，那么每个神经元就会计算出相同的结果，在反向传播的时候也会计算出相同的梯度，最后导致所有权重都会有相同的更新。换句话说，如果每个权重都被初始化成相同的值，那么权重之间失去了不对称性。

2. 随机初始化

目前知道我们希望权重初始化的时候能够尽量靠近 0，但是不能全都等于 0，所以可以初始化权重为一些靠近 0 的随机数，通过这个方式可以打破对称性。这里面的核心想法就是神经元最开始都是随机的、唯一的，所以在更新的时候也是作为独立的部分，最后一起合成在神经网络当中。

一般的随机化策略有高斯随机化、均匀随机化等，需要注意的是并不是越小的随机化产生的结果越好，因为权重初始化越小，反向传播中关于权重的梯度也越小，因为梯度与参数的大小是成比例的，所以这会极大地减弱梯度流的信号，成为神经网络训练中的一个隐患。

这个初始化策略还存在一个问题就是网络输出分布的方差会随着输入维度的增加而增大，可以用下面的数学式子来说明：

$$Var(s) = Var(\sum_{i=1}^{n} w_i x_i) \tag{3.60}$$

$$= \sum_{i=1}^{n} Var(w_i x_i) \tag{3.61}$$

$$= \sum_{i=1}^{n} [E(w_i)]^2 Var(x_i) + E[(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i) \tag{3.62}$$

$$= \sum_{i=1}^{n} Var(x_i)Var(w_i) \tag{3.63}$$

$$= (nVar(w))Var(x) \tag{3.64}$$

其中假设输入和权重都是 0 均值的，也就是说 $E[x_i] = E[w_i] = 0$，但这并不是一般的情况，比如经过了 ReLU 激活函数之后输出就会是一个正的均值，这里做这样的假设是为了方便计算。我们可以看到输出的结果 S 比输入 x 的方差增大了 $nVar(w)$ 倍，如果网络越来越深，就会导致方差越来越大，所以希望 $nVar(w)$ 尽可能接近 1，也就是 $nVar(w) = 1$，这可以得到 $Var(w) = \frac{1}{n}$，也就是 w 初始化之后需要除以 $\sqrt{n}$。

3. 稀疏初始化

另外一种初始化的方法就是稀疏初始化，将权重全部初始化为 0，然后为了打破对称性在里面随机挑选一些参数附上一些随机值。这种方法的好处是参数占用的内存较少，因为里面有较多的 0，但是实际中使用较少。

4. 初始化偏置（bias）

对于偏置 (bias)，通常是初始化为 0，因为权重已经打破了对称性，所以使用 0 来初始化是最简单的。

5. 批标准化（Batch Normalization）

最近兴起的一项技术叫做批标准化，它的核心想法就是标准化这个过程是可微的，减少了很多不合理初始化的问题，所以我们可以将标准化过程应用到神经网络的每一层中做前向传播和反向传播，通常批标准化应用在全连接层后面、非线性层前面。

实际中批标准化已经变成了神经网络中的一个标准技术，特别是在卷积神经网络中，它对于很坏的初始化有很强的鲁棒性，同时还可以加快网络的收敛速度。另外，批标准化还可以理解为在网络的每一层前面都会做数据的预处理。

3.7.3 防止过拟合

之前在 3.4.5 节中我们讲到了如果网络容量过大会造成过拟合，但是防止过拟合的最佳办法并不是减少网络容量，那么如何防止过拟合呢？下面我们会讲几个具体的办法来防止过拟合。

1. 正则化

L2 正则化是正则化（regularization）中比较常用的形式，它的想法是对于权重过大的部分进行惩罚，也就是直接在损失函数中增加权重的二范数量级，也就是 $\frac{1}{2}\lambda w^2$，其中 λ 是正则化强度，通常使用 0.5，因为对于 w^2 的梯度是 $2w$，使用 $\frac{1}{2}$ 就能使得梯度是 λw 而不是 $2\lambda w$。所以使用 L2 正则化可以看成是权重更新在原来的基础上再 $-\lambda w$，这样可以让参数更新之后更加靠近 0。

L1 正则化是另外一种正则化方法，其在损失函数中增加权重的 1 范数，也就是 $\lambda|w|$，我们也可以把 L1 正则化和 L2 正则化结合起来，如 $\lambda_1|w| + \lambda_2 w^2$。L1 正则化相对于 L2 正则化的优势是在优化的过程中可以让权重变得更加稀疏，换句话说，也就是在优化结束的时候，权重只会取一些与最重要的输入有关的权重，这就使得与噪声相关的权重被尽可能降为 0。L2 正则化的优势在于最终的效果会比 L1 正则化更加发散，权重也会被限制得更小。

除此之外，还有一种正则化方法叫做最大范数限制，其迫使权重在更新的过程中范数有一个上界，也就是 $||w|| < c$，这种办法可以使得当学习率设置太高的时候网络不会“爆炸”，因为更新总是有界的。

在实际中对于正则化的选择通常使用 L2 正则化，其使用更加常见。

2.Dropout

现在介绍一种非常有效、简单、同时也是现在深度学习使用最为广泛的防止过拟合的方法——Dropout。其核心想法就是在训练网络的时候依概率 P 保留每个神经元，也就是说每次训练的时候有些神经元会被设置为 0，其简单示意图如 3.26所示。

通过图 3.26 我们可以看到每次训练都有某些神经元并没有参与到网络中，但是在预测的时候不再这样处理，这也很好理解，如果预测应用 Dropout，由于随机性，每次预测出来的结果都不一样，这样预测的时候完全靠运气，这显然是不行的。所以会保留网络全部的权重，取代应用 Dropout，在每层网络的输出上应用 P 的缩放。这个想法是很重要的，因为如果我们不做任何处理，那么网络的行为在预测时和训练时就会不同，这不是所希望的，所以需要应用缩放。

那么为什么在网络输出部分应用 P 缩放可以达到相同的效果呢？考虑一个神经元在应用 Dropout 之前的输出是 x，那么应用 Dropout 之后它的输出期望值就是 $Px + (1 - P)0$，所以在预测的时候，如果保留所有的权重，就必须调整 $x \to Px$ 来保证其输出与期待相同。

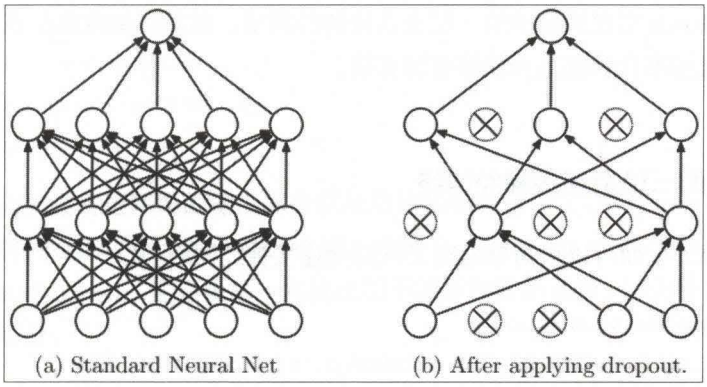


图 3.26 简单神经元

还有一种理解 Dropout 的思路就是把 Dropout 看作是集成的学习方法，如图3.27所示。

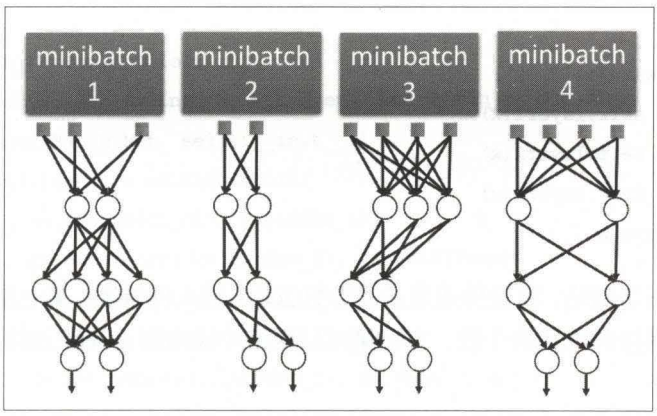


图 3.27 集成的学习方法

每一次训练 Dropout 之后就可以看作是一个新的模型，然后训练了很多次之后就可以看成是这些模型的集成。

上面介绍了多种防止过拟合的办法，在实际应用中，一般采用全局权重的 L2 正则化搭配 Dropout 来防止过拟合。

3.8 多层全连接神经网络实现 MNIST 手写数字分类

前面我们已经讲完了很多理论知识，“Talk is cheap，show me the code”，是时候来实践一下我们的算法了。

先用 PyTorch 实现最简单的三层全连接神经网络，然后添加激活层查看试验结果，最后再加上批标准化验证是否能够更加有效。

3.8.1 简单的三层全连接神经网络

在 PyTorch 里面可以很简单地定义三层全连接神经网络。

```
class simpleNet(nn.Module):
    def __init__(self, in_dim, n_hidden_1, n_hidden_2, out_dim):
        super(simpleNet, self).__init__()
        self.layer1 = nn.Linear(in_dim, n_hidden_1)
        self.layer2 = nn.Linear(n_hidden_1, n_hidden_2)
        self.layer3 = nn.Linear(n_hidden_2, out_dim)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

对于这个三层网络，需要传递进去的参数包括：输入的维度、第一层网络的神经元个数、第二层网络神经元的个数，以及第三层网络（输出层）神经元的个数。

3.8.2 添加激活函数

接着改进一下网络，添加激活函数增加网络的非线性，方法也非常简单。

```
class Activation_Net(nn.Module):
    def __init__(self, in_dim, n_hidden_1, n_hidden_2, out_dim):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Linear(in_dim, n_hidden_1), nn.ReLU(True))
        self.layer2 = nn.Sequential(
            nn.Linear(n_hidden_1, n_hidden_2), nn.ReLU(True))
        self.layer3 = nn.Sequential(nn.Linear(n_hidden_2, out_dim))

    def forward(self, x):
```

```
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

这里只需要在每层网络的输出部分添加激活函数就可以了，用到 `nn.Sequential()`，这个函数是将网络的层组合到一起，比如上面将 `nn.Linear()` 和 `nn.ReLU()` 组合到一起作为 `self.layer`。注意最后一层输出层不能添加激活函数，因为输出的结果表示的是实际的得分。

3.8.3 添加批标准化

最后添加一个加快收敛速度的方法——批标准化。

```
class Batch_Net(nn.Module):
    def __init__(self, in_dim, n_hidden_1, n_hidden_2, out_dim):
        super(Batch_Net, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Linear(in_dim, n_hidden_1),
            nn.BatchNorm1d(n_hidden_1), nn.ReLU(True))
        self.layer2 = nn.Sequential(
            nn.Linear(n_hidden_1, n_hidden_2),
            nn.BatchNorm1d(n_hidden_2), nn.ReLU(True))
        self.layer3 = nn.Sequential(nn.Linear(n_hidden_2, out_dim))

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

同样使用 `nn.Sequential()` 将 `nn.BatchNorm1d()` 组合到网络层中，注意批标准化一般放在全连接层的后面、非线性层（激活函数）的前面。

3.8.4 训练网络

网络的定义特别简单，现在用 MNIST 数据集训练网络并测试一下每种网络的结果。

MNIST 数据集是一个手写字体数据集，包含 0 到 9 这 10 个数字，其中有 55000 张训练集，10000 张测试集，5000 张验证集，图片大小是 28×28 的灰度图，如图 3.28所示。

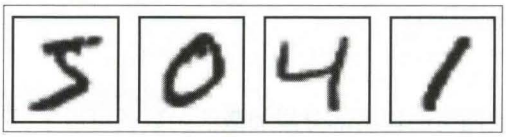


图 3.28　训练图序

首先需要导入一些要用的包：

```
import torch
from torch import nn, optim
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

import net
```

最下面的 `import net` 是之前定义网络的 Python 文件。

然后可以定义一些超参数，如 `batch_size`、`learning_rate` 还有 `num_epoches` 等。

```
# 超参数(Hyperparameters)
batch_size = 64
learning_rate = 1e-2
num_epoches = 20
```

接着需要进行数据预处理，就像之前介绍的，需要将数据标准化，这里运用到的函数是 `torchvision.transforms`，它提供了很多图片预处理的方法。这里使用两个方法：第一个是 `transforms.ToTensor()`，第二个是 `transforms.Normalize()`。

`transform.ToTensor()` 很好理解，就是将图片转换成 PyTorch 中处理的对象 Tensor，在转化的过程中 PyTorch 自动将图片标准化了，也就是说 Tensor 的范围是 0 ～ 1。接着我们使用 `transforms.Normalize()`，需要传入两个参数：第一个参数是均值，第二个参数是方差，做的处理就是减均值，再除以方差。

```
data_tf = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize([0.5], [0.5])])
```

这里 transforms.Compose() 将各种预处理操作组合到一起，transforms.Normalize([0.5], [0.5]) 表示减去 0.5 再除以 0.5，这样将图片转化到了 −1 ～ 1 之间，注意因为图片是灰度图，所以只有一个通道，如果是彩色的图片，有三通道，那么用 transforms.Normali ze([a, b, c], [d, e, f]) 来表示每个通道对应的均值和方差。

然后读取数据集。

```
# 下载训练集 MNIST 手写数字训练集
train_dataset = datasets.MNIST(
    root='./data', train=True, transform=data_tf, download=True)

test_dataset = datasets.MNIST(root='./data', train=False, transform=data_tf)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

通过 PyTorch 的内置函数 torchvision.datasets.MNIST 导入数据集，传入数据预处理，前面介绍了如何定义自己的数据集，之后会用具体的例子说明。接着使用 torch.utils.data.DataLoader 建立一个数据迭代器，传入数据集和 batch_size，通过 shuffle=True 来表示每次迭代数据的时候是否将数据打乱。

接着导入网络，定义损害函数和优化方法。

```
model = net.simpleNet(28 * 28, 300, 100, 10)
if torch.cuda.is_available():
    model = model.cuda()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

net.simpleNet 是定义的简单三层网络，里面的参数是 $28\times28, 300, 100, 10$，其中输入的维度是 28×28，因为输入图片大小是 28×28，然后定义两个隐藏层分别是 300 和 100，最后输出的结果必须是 10，因为这是一个分类问题，一共有 0 ～ 9 这 10 个数字，所以是 10 分类。损失函数定义为分类问题中最常见的损失函数交叉熵，使用随机梯度下降来优化损失函数。

接着开始训练网络，流程基本和之前一致，这里就不再赘述。最后训练完网络之后需要测试网络，通过下面的代码来测试。

```
model.eval()
eval_loss = 0
eval_acc = 0
for data in test_loader:
    img, label = data
    img = img.view(img.size(0), -1)
    if torch.cuda.is_available():
        img = Variable(img, volatile=True).cuda()
        label = Variable(label, volatile=True).cuda()
    else:
        img = Variabel(img, volatile=True)
        label = Variable(label, volatile=True)
    out = model(img)
    loss = criterion(out, label)
    eval_loss += loss.data[0] * label.size(0)
    _, pred = torch.max(out, 1)
    num_correct = (pred == label).sum()
    eval_acc += num_correct.data[0]
print('Test Loss: {:.6f}, Acc: {:.6f}'.format(
    eval_loss / (len(test_dataset)),
    val_acc / (len(test_dataset))))
```

这里需要注意的一点就是 `img = Variable(img, volatile=True)`,里面的 `volatile=True` 表示前向传播时不会保留缓存，因为对于测试集，不需要做反向传播，所以可以在前向传播时释放掉内存，节约内存空间。

最后看一下每种网络的测试结果，如图 3.29、图3.30、图3.31所示。

图 3.29、图 3.30 和图 3.31 依次是简单三层网络、添加激活层的网络和添加了批标准化的网络，可以从中看出网络的准确率越来越高，这也证明了之前所介绍的内容的正确性，同时还可以进入网络中增加网络的泛化能力，如 Dropout、正则化等，还可以修改隐藏层的神经元个数，增加隐藏层层数等，这里留给读者自己做实验去练习。

```
epoch 20
**********
[300/938] Loss: 0.262563, Acc: 0.924948
[600/938] Loss: 0.267500, Acc: 0.925078
[900/938] Loss: 0.268621, Acc: 0.924826
Finish 20 epoch, Loss: 0.267833, Acc: 0.925133
Test Loss: 0.277661, Acc: 0.919800
```

图 3.29　简单三层网络测试结果

```
epoch 20
**********
[300/938] Loss: 0.087531, Acc: 0.975521
[600/938] Loss: 0.089605, Acc: 0.974010
[900/938] Loss: 0.089114, Acc: 0.974149
Finish 20 epoch, Loss: 0.088823, Acc: 0.974267
Test Loss: 0.100920, Acc: 0.968800
```

图 3.30 添加激活层的网络测试结果

```
epoch 20
**********
[300/938] Loss: 0.016224, Acc: 0.996927
[600/938] Loss: 0.016665, Acc: 0.996641
[900/938] Loss: 0.017092, Acc: 0.996632
Finish 20 epoch, Loss: 0.017219, Acc: 0.996567
Test Loss: 0.061574, Acc: 0.981100
```

图 3.31 添加类比标准化的网络测试结果

至此便介绍完简单的多层全连接神经网络，最后也用 PyTorch 实现了不同结构的网络，通过在 MNIST 数据集上的训练，比较了它们之间的差异。

下一章将正式进入时下最流行的深度学习领域，从计算机视觉入手，引入特别适合做图像处理的网络模型——卷积神经网络。

第 4 章

卷积神经网络

图像分类问题是计算机视觉中的一个核心问题，虽然问题描述很简单，却有着很广泛的实用价值，很多独立的计算机视觉任务如目标检测、分割等，都可以简化为图像分类问题。卷积神经网络于 1998 年由 Yann Lecun 提出。2012 年，Alex Krizhevsky 凭借它赢得了 ImageNet 挑战赛，震惊世界，如今卷积神经网络已经成为计算机视觉领域最具影响力的一部分。

这一章将从计算机视觉的任务起源开始引入卷积神经网络，介绍卷积神经网络的原理和基础，然后介绍 PyTorch 的卷积模块，接着介绍现在应用最广泛的几个卷积神经网络模型，然后用卷积神经网络再次实现上一章的 MNIST 手写数字分类，并介绍图像增强的技巧，运用它在新的数据集 cifar10 上进行测试，最后介绍卷积神经网络的逆过程：反卷积神经网络。

4.1 主要任务及起源

人类获取外界信息，主要依靠视觉、听觉、触觉、嗅觉和味觉等感觉器官，其中 80% 的信息获取都来自视觉，而且视觉获取的信息也是最丰富、最复杂的。人的生理构造决定了我们能够看清楚并理解身边的场景，而要让计算机看懂这个世界却是一件非常困难的事情，即便在很多人看来现在的计算机技术已经足够先进了，但是要达到看懂并自主分析各种复杂信息的程度，还有很长的一段路要走，这也是计算机视觉这门学科要解决的事情。

计算机视觉的核心任务之一是图像识别，我们总是在想能不能教会机器来完成图像识别这个任务。人类对于图片的识别相当容易，然而机器却面临了很多问题，如视角变换、光照条件、背景干扰、物体变形等，正是由于这些问题的干扰使得计算机在识别图片的时候准确率总是太低。

如何写一个算法来分类图片呢？和排序不同，分类图片并没有那么简单，我们不可能自己制定一个规则决定哪张图片属于哪一类，所以需要通过学习算法让机器自己知道如何分类。我们将会给计算机提供每种类别的图片，让机器自己去学习其中的特征并形成一个算法，这就是机器学习的核心。这些算法是依赖于数据集的，所以也称为数据驱动的算法。

在卷积神经网络流行起来之前，图像处理使用的都是一些传统的方法，比如提取图像中的边缘、纹理、线条、边界等特征，依据这些特征再进行下一步的处理，这样的处理不仅效率特别低，准确率也不高。随着计算机视觉的快速发展，如今在某些图像集上机器的识别准确率已经超过了人类，这一切都要归功于卷积神经网络。

下面就来介绍什么是卷积神经网络。

4.2 卷积神经网络的原理和结构

在介绍卷积神经网络之前，先提出三个观点，正是这三个观点使得卷积神经网络能够真正起作用。

1. 局部性

对于一张图片而言，需要检测图片中的特征来决定图片的类别，通常情况下这些特征都不是由整张图片决定的，而是由一些局部的区域决定的。

比如图4.1中的鸟喙，该特征只存在于图片的局部中。

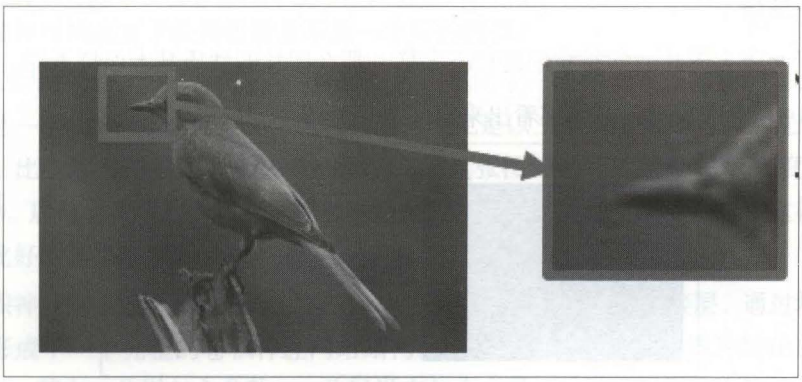


图 4.1 图片特征只在局部

2. 相同性

对于不同的图片，如果它们具有同样的特征，这些特征会出现在图片不同的位置，也就是说可以用同样的检测模式去检测不同图片的相同特征，只不过这些特征处于图片中不同的位置，但是特征检测所做的操作几乎一样。

图 4.2中两张图片的鸟喙处于不同的位置，但是可以用相同的检测模式去检测。

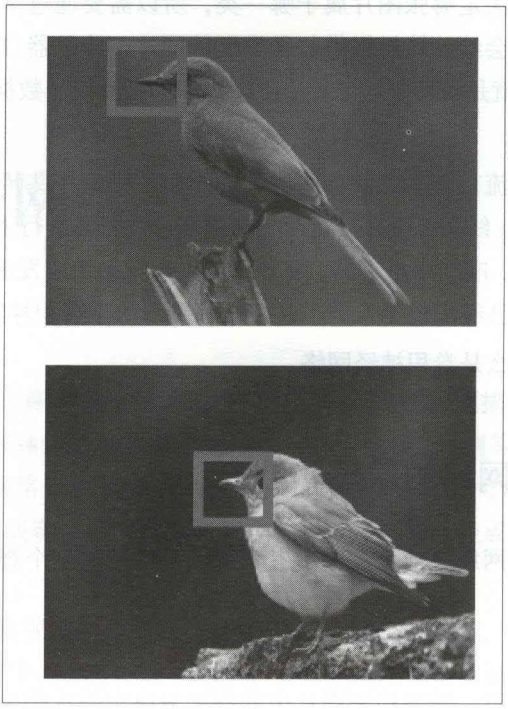


图 4.2　图片特征在不同的位置

3. 不变性

对于一张大图片，如果我们进行下采样，那么图片的性质基本保持不变。

图 4.3经过下采样还是能够看出来是一张鸟的图片。

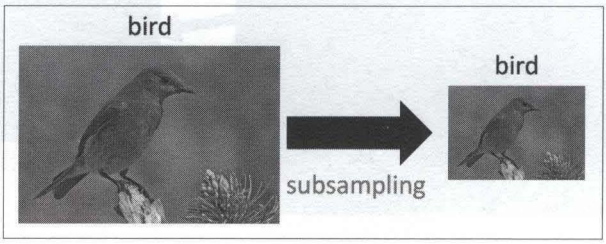


图 4.3　下采样

上面的三个性质分别对应着卷积神经网络中的三种思想，接下来介绍网络的层结构。

第 3 章介绍了一般的全连接神经网络，卷积神经网络和它是相似的，也是由一些神经元构成的，如图 4.4 所示。这些神经元中有着需要学习的参数，通过网络输入，最后输出结果，通过损失函数来优化网络中的参数。卷积神经网络与其不同之处在于网络的层结构是不同的。图4.4是全连接神经网络，由一系列隐藏层构成，每个隐藏层由若干个神经元构成，其中每个神经元都和前一层的所有神经元相关联，但是每一层中的神经元是相互独立的。

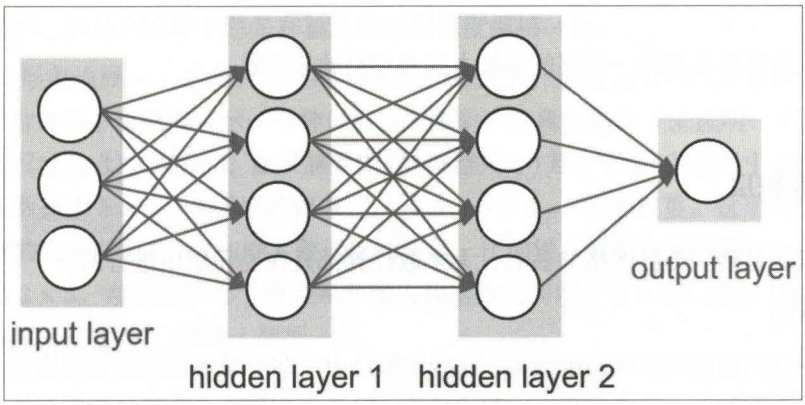


图 4.4 全连接神经网络

这样的神经网络在处理图片上存在什么问题呢？比如在 MNIST 数据集上，图片大小是 28×28，那么第一个隐藏层的单个神经元的权重数目就是 $28 \times 28 = 784$，这似乎还不是特别大，但这只是一张小图片，且是灰度图。对于一张较大的图片而言，比如 $200 \times 200 \times 3$，就会导致权重数目是 $200 \times 200 \times 3 = 120000$，如果设置几个隐藏层中的神经元数目，就会导致参数增加特别快。其实这样的图片在现实中并不算大图片，所以全连接神经网络对于处理图像并不是一个好的选择。

图4.5所示的是卷积神经网络的处理过程，不同于一般的全连接神经网络，卷积神经网络是一个 3D 容量的神经元，也就是说神经元是以三个维度来排列的：宽度、高度和深度。比如输入的图片是 $32 \times 32 \times 3$，那么这张图片的宽度就是 32，高度也是 32，深度是 3。后面会详细地介绍卷积神经网络是如何计算的，以及为什么它被这样设计并取得如此好的效果。

卷积神经网络中的主要层结构有三个：卷积层、池化层和全连接层，通过堆叠这些层结构形成了一个完整的卷积神经网络结构。卷积神经网络将原始图片转化成最后的类别得分，其中一些层包含参数，一些层没有包含参数，比如卷积层和全连接层拥有参数，而激活层和池化层不含参数。这些参数通过梯度下降法来更新，最后使得模型尽可

能正确地识别出图片类别。

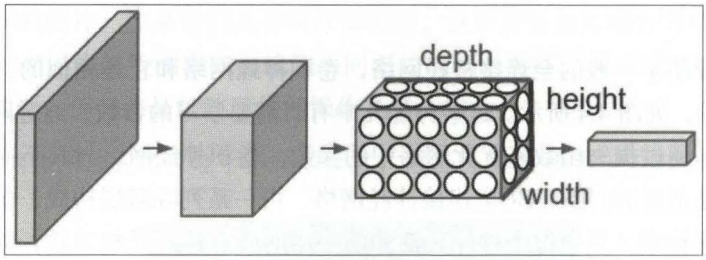


图 4.5　卷积神经网络的处理过程

接下来具体介绍每一种层的连接方式和它们的超参数。

4.2.1　卷积层

卷积层是卷积神经网络的核心，大多数计算都是在卷积层中进行的。

1. 概述

首先介绍卷积神经网络的参数。这些参数是由一些可学习的滤波器集合构成的，每个滤波器在空间上（宽度和高度）都比较小，但是深度和输入数据的深度保持一致。举例来说，卷积神经网络的第一层卷积一个典型的滤波器的尺寸可以是 $5 \times 5 \times 3$（宽和高都是 5），或者是 $3 \times 3 \times 3$（宽和高都是 3），这里的宽度和高度可以任意定义，但是深度必须是 3，因为深度要和输入一致，而输入的图片是 3 通道的。在前向传播的时候，让每个滤波器都在输入数据的宽度和高度上滑动（卷积），然后计算整个滤波器和输入数据任意一处的内积。

当滤波器沿着输入数据的宽度和高度滑动时，会生成一个二维的激活图，激活图上的每个空间位置表示了原图片对于该滤波器的反应。直观来看，网络会让滤波器学习到当它看到某些类型的视觉特征的时候就激活，具体的视觉特征可以是边界、颜色、轮廓、甚至可以是网络更高层上的蜂巢状或者车轮状图案。

在每个卷积层上，会有一整个集合的滤波器，比如 20 个，这样就会形成 20 张二维的、不同的激活图，将这些激活图在深度方向上层叠起来就形成了卷积层的输出。

如果用大脑和生物神经元做比喻，那么输出的 3D 数据中的每个数据都可以看成是神经元的一个输出，而该神经元只是观察输入数据中的一种特征，并且和空间上左右两边的所有神经元共享参数（因为这些输出都是使用同一个滤波器得到的结果）。下面介绍卷积神经网络中的神经元连接，它们在空间中的排列，以及它们参数共享的模式。

2. 局部连接

在处理图像这样高维度输入的时候，让每个神经元都与它那一层中的所有神经元进行全连接是不现实的。相反，让每个神经元只与输入数据的一个局部区域连接是可行的，为什么可以这样做呢？其实这是因为图片特征的局部性，所以只需要通过局部就能提取出相应的特征。

与神经元连接的空间大小叫做神经元的感受野（receptive field），它的大小是一个人为设置的超参数，这其实就是滤波器的宽和高。在深度方向上，其大小总是和输入的深度相等。最后强调一下，对待空间维度（宽和高）和深度维度是不同的，连接在空间上是局部的，但是在深度上总是和输入的数据深度保持一致。

图4.6形象地展示了感受野在空间和深度上的大小，左边表示输入的数据，中间是感受野，右边每个小圆点表示一个神经元。下面举一个具体的例子来说明一下。比如输入的数据尺寸为 $32 \times 32 \times 3$，如果感受野 (滤波器尺寸) 是 5×5，卷积层中每个神经元会有输入数据中 $5 \times 5 \times 3$ 区域的权重，一共 $5 \times 5 \times 3 = 75$ 个权重。这里再次强调感受野深度的大小必须是 3，和输入数据保持一致。比如输入数据体尺寸是 $16 \times 16 \times 20$，感受野是 3×3，卷积层中每个神经元和输入数据体之间就有 $3 \times 3 \times 20 = 180$ 个连接，这里的深度必须是 20，和输入数据一致。

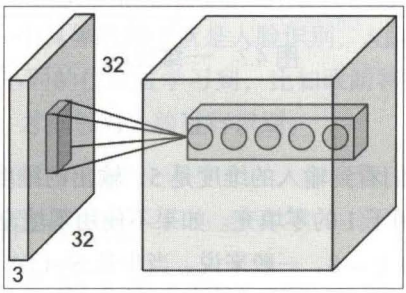


图 4.6　感受野

3. 空间排列

前面介绍了每个神经元只需要与输入数据的局部区域相连接，但是没有介绍卷积层中神经元的数量和它们的排列方式、输出深度、滑动步长，以及边界填充控制着卷积层的空间排布。

首先，卷积层的输出深度是一个超参数，它与使用的滤波器数量一致，每种滤波器所做的就是在输入数据中寻找一种特征。比如说输入一张原始图片，卷积层输出的深度是 20，这说明有 20 个滤波器对数据进行处理，每种滤波器寻找一种特征进行激活。

其次，在滑动滤波器的时候，必须指定步长。比如步长为 1，说明滤波器每次移动 1 个像素点。当步长为 2 的时候，滤波器会滑动 2 个像素点。滑动的操作会使得输出的

数据在空间上变得更小。

最后介绍边界填充，可以将输入数据用 0 在边界进行填充，这里将 0 填充的尺寸作为一个超参数，有一个好处就是，可以控制输出数据在空间上的尺寸，最常用来保证输入和输出在空间上尺寸一致。

输出的尺寸到底是多少呢？其实可以用一个公式来计算，就是 $\frac{W-F+2P}{S}+1$，其中 W 表示输入的数据大小，F 表示卷积层中神经元的感受野尺寸，S 表示步长，P 表示边界填充 0 的数量。比如输入是 7×7，滤波器是 3×3，步长是 1，填充的数量是 0，那么根据公式，就能得到 $\frac{7-3+2\times 0}{1}+1=5$，即输出的空间大小是 5×5，如果步长是 2，那么 $\frac{7-3+2\times 0}{2}+1=3$，输出的空间大小就是 3×3。可以用图 4.7所示的这个一维的例子来具体说明。

右上角表示神经网络的权重，其中输入数据的大小为 5，感受野的大小为 3；左边表示滑动步长为 1，且填充也为 1；右边表示滑动步长为 2，填充为 1。

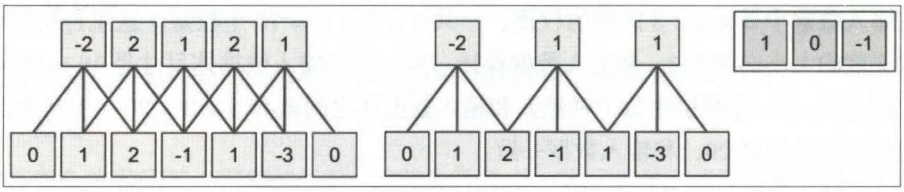


图 4.7　一维空间

4. 零填充的使用

从上面的例子中，我们看到输入的维度是 5，输出的维度也是 5。之所以如此，是因为感受野是 3，并且使用了 1 的零填充。如果不使用零填充，那么输出数据的维度也就只有 3，因为 $\frac{5-3+2\times 0}{1}+1=3$。一般来说，当步长 S=1 时，零填充的值为 $P=\frac{F-1}{2}$，这样就能够保证输入的数据和输出的数据具有相同的空间尺寸。

5. 步长的限制

通过上面的公式我们知道步长的选择是有所限制的，举例来说，当输入尺寸 $W=10$ 的时候，如果不使用零填充，即 $P=0$，滤波器尺寸 $F=3$，这样步长 $S=2$ 就行不通，因为 $\frac{10-3+0}{2}+1=4.5$，结果不是一个整数，这就说明神经元不能整齐对称地滑过输入数据体，这样的超参数设置是无效的，使用 PyTorch 的时候就会报错，可以使用零填充让设置变得合理。在后面卷积神经网络的结构设计中，需要合理地设计网络的尺寸，使得所有维度都能够正常工作，这件事并没有看上去那么容易。

6. 参数共享

在卷积层使用参数共享可以有效地减少参数的个数，这样之所以能够行得通，是因为之前介绍的特征的相同性，也就是说相同的滤波器能够检测出不同位置的相同特

征。比如说一个卷积层的输出是 $20 \times 20 \times 32$，那么其中神经元的个数就是 $20 \times 20 \times 32 = 12800$，如果窗口大小是 3×3，而输入的数据体深度是 10，那么每个神经元就有 $3 \times 3 \times 10 = 900$ 个参数，这样合起来就有 $12800 \times 900 = 11520000$ 个参数，单单一层卷积就有这么多参数，这样运算速度显然是特别慢的。

根据之前介绍的，一个滤波器检测出一个空间位置 (x_1, y_1) 处的特征，那么也能够有效检测出 (x_2, y_2) 位置的特征，所以就可以用相同的滤波器来检测相同的特征，基于这个假设，我们就能够有效减少参数个数。比如上面这个例子，一共有 32 个滤波器，这使得输出体的厚度是 32，每个滤波器的参数为 $3 \times 3 \times 10 = 900$，总共的参数就有 $32 \times 900 = 28800$ 个，极大减少了参数的个数。

由参数共享我们知道输出体数据在深度切片上所有的权重都使用同一个权重向量，那么卷积层在向前传播的过程中，每个深度切片都可以看成是神经元的权重对输入数据体做卷积，这也就是为什么要把这些 3D 的权重集合称为滤波器，或者卷积核。

需要注意的是，参数共享之所以能够有效，是因为一个特征在不同位置的表现是相同的，比如一个滤波器检测到了水平边界这个特征，那么这个特征具有平移不变性，所以在其他位置也能够检测出来。但是有时候这样的假设可能是没有意义的，特别是当卷积神经网络的输入图像呈现的是一些明确的中心结构的时候，希望在图片的不同位置学习到不同的特征。一个具体的例子就是人脸识别，人脸一般位于图片的中心，我们希望不同的特征能够在不同的位置被学习到，比如眼睛特征或者头发特征，正是由于这些特征在不同的地方，才能够对人脸进行识别。

7. 总结

最后总结一下卷积层的一些性质。

（1）输入数据体的尺寸是 $W_1 \times H_1 \times D_1$。

（2）4 个超参数：滤波器数量 K，滤波器空间尺寸 F，滑动步长 S，零填充的数量 P。

（3）输出数据体的尺寸为 $W_2 \times H_2 \times D_2$，其中 $W_2 = \frac{W_1 - F + 2P}{S} + 1$，$H_2 = \frac{H_1 - F + 2P}{S} + 1$，$D_2 = K$。

（4）由于参数共享，每个滤波器包含的权重数目为 $F \times F \times D_1$，卷积层一共有 $F \times F \times D_1 \times K$ 个权重和 K 个偏置。

（5）在输出体数据中，第 d 个深度切片（空间尺寸是 $W_2 \times H_2$），用第 d 个滤波器和输入数据进行有效卷积运算的结果，再加上第 d 个偏置。

对于卷积神经网络的一些超参数，常见的设置是 $F = 3$，$S = 1$，$P = 1$，同时这些超参数也有一些约定俗成的惯例和经验，在之后的章节会介绍。

4.2.2 池化层

上一部分介绍完卷积神经网络中最核心的内容——卷积层，下面来介绍一下第二种层结构——池化层。

通常会在卷积层之间周期性插入一个池化层，其作用是逐渐降低数据体的空间尺寸，这样就能够减少网络中参数的数量，减少计算资源耗费，同时也能够有效地控制过拟合。

下面先来介绍到底什么是池化层。池化层和卷积层一样也有一个空间窗口，通常采用的是取这些窗口中的最大值作为输出结果，然后不断滑动窗口，对输入数据体每一个深度切片单独处理，减少它的空间尺寸，如图4.8所示。

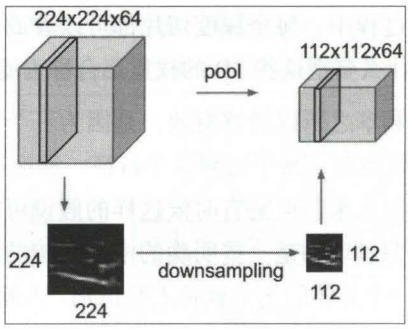


图 4.8　池化层处理效果

从图4.8能够看出池化层能够有效降低数据体空间的大小，图4.9形象地说明了窗口大小是 2，滑动步长是 2 的最大值池化是如何计算的：每次都从 2 × 2 的窗口中选择最大的数值，同时每次滑动 2 个步长进入新的窗口。

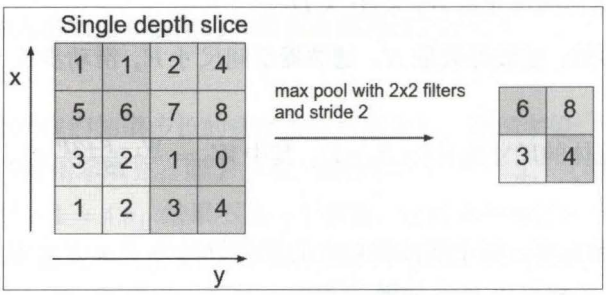


图 4.9　池化层计算

池化层之所以有效，是因为之前介绍的图片特征具有不变性，也就是通过下采样不会丢失图片拥有的特征，由于这种特性，我们可以将图片缩小再进行卷积处理，这样能够大大降低卷积运算的时间。

最常用的池化层形式是尺寸为 2×2 的窗口，滑动步长为 2，对图像进行下采样，将其中 75% 的激活信息都丢掉，选择其中最大的保留下来，这其实是因为我们希望能够更加激活里面的数值大的特征，去除一些噪声信息。

池化层有一些和卷积层类似的性质。

（1）输入数据体的尺寸是 $W_1 \times H_1 \times D_1$。

（2）有两个需要设置的超参数，空间大小 F 和滑动步长 S。

（3）输出数据体的尺寸是 $W_2 \times H_2 \times D_2$，其中 $W_2 = \frac{W_1 - F}{S} + 1, H_2 = \frac{H_1 - F}{S} + 1, D_2 = D_1$。

（4）对输入进行固定函数的计算，没有参数引入。

（5）池化层中很少引入零填充。

在实际中，有两种方式：一种是 $F = 3, S = 2$，这种池化有重叠；另外更常用的一种是 $F = 2, S = 2$，一般来说应该谨慎使用比较大的池化窗口，以免对网络有破坏性。

除了最大值池化之外，还有一些其他的池化函数，比如平均池化，或者 L2 范数池化。在实际中证明，在卷积层之间引入最大池化的效果是最好的，而平均池化一般放在卷积神经网络的最后一层。

4.2.3 全连接层

全连接层和之前介绍的一般的神经网络的结构是一样的，每个神经元与前一层所有的神经元全部连接，而卷积神经网络只和输入数据中的一个局部区域连接，并且输出的神经元每个深度切片共享参数。

一般经过了一系列的卷积层和池化层之后，提取出图片的特征图，比如说特征图的大小是 $3 \times 3 \times 512$，这个时候，将特征图中的所有神经元变成全连接层的样子，直观上也就是将一个 3D 的立方体重新排列，变成一个全连接层，里面有 $3 \times 3 \times 512 = 4608$ 个神经元，再经过几个隐藏层，最后输出结果。

在这个过程中为了防止过拟合会引入 Dropout。最近的研究表明，在进入全连接层之前，使用全局平均池化能够有效地降低过拟合。

4.2.4 卷积神经网络的基本形式

卷积神经网络中通常是由上面介绍的三种层结构所构成，上一章还介绍过引入激活函数增加模型的非线性，所以卷积神经网络最常见的形式就是将一些卷积层和 ReLU

层放在一起，有可能在 ReLU 层前面加上批标准化层，随后紧跟着池化层，再不断重复，直到图像在空间上被缩小到一个足够小的尺寸，然后将特征图展开，连接几层全连接层，最后输出结果，比如分类评分等。

图 4.10就是一种卷积神经网络的基本形式。

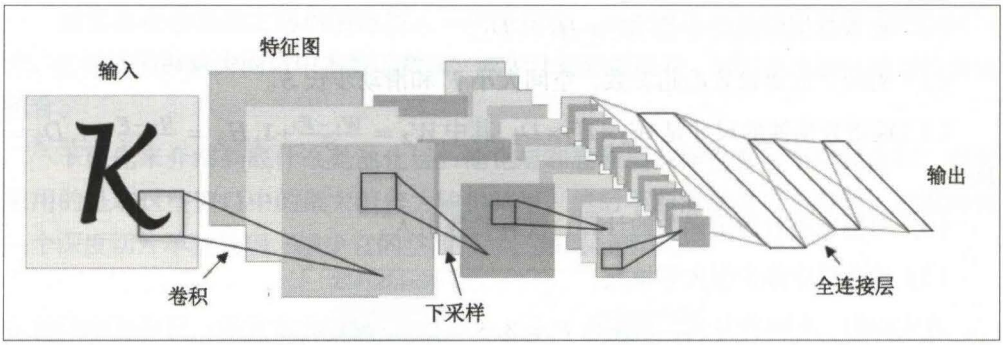


图 4.10　卷积神经网络的基本形式

1. 小滤波器的有效性

一般而言，几个小滤波器卷积层的组合比一个大滤波器卷积层要好，比如层层堆叠了 3 个 3×3 的卷积层，中间含有非线性激活层，在这种排列下面，第一个卷积层中每个神经元对输入数据的感受野是 3×3，第二层卷积层对第一层卷积层的感受野也是 3×3，这样对于输入数据的感受野就是 5×5，同样，第三层卷积层上对第二层卷积层的感受野是 3×3，这样第三层卷积层对于第一层输入数据的感受野就是 7×7。

假设这里不使用 3 个 3×3 的感受野，直接单独使用一个 7×7 大小的卷积层，那么所有神经元的感受野也是 7×7，但是这样会有一些缺点。多个卷积层首先与非线性激活层交替的结构，比单一卷积层的结构更能提取出深层的特征；其次，假设输入数据体的深度是 C，输出体的深度也是 C，那么单独的 7×7 的卷积层会有 $7 \times 7 \times C \times C = 49 \times C^2$ 的参数个数，而使用 3 个 3×3 的卷积层的组合，仅仅含有 $3 \times (3 \times 3 \times C \times C) = 27 \times C^2$ 的参数。直观来说，选择小滤波器的卷积组合能够对输入数据表达出更有力的特征，同时使用参数也更少。唯一的不足是反向传播更新参数的时候，中间的卷积层可能会占用更多的内存。

2. 网络的尺寸

对于卷积神经网络的尺寸设计，没有严格的数学证明，这是根据经验制定出来的规则。

（1）输入层：一般而言，输入层的大小应该能够被 2 整除很多次，常用的数字包括 32，64，96 和 224。

（2）卷积层：卷积层应该尽可能使用小尺寸的滤波器，比如 3×3 或者 5×5，滑动步长取 1。还有一点就是需要对输入数据体进行零填充，这样可以有效地保证卷积层不会改变输入数据体的空间尺寸。如果必须要使用更大的滤波器尺寸，比如 7×7，通常用在第一个面对原始图像的卷积层上。

（3）池化层：池化层负责对输入的数据空间维度进行下采样，常用的设置使用 2×2 的感受野做最大值池化，滑动步长取 2。另外一个不常用的设置是使用 3×3 的感受野，步长设置为 2。一般而言池化层的感受野大小很少超过 3，因为这样会使得池化过程过于激烈，造成信息的丢失，这通常会造成算法的性能变差。

（4）零填充：零填充的使用可以让卷积层的输入和输出在空间上的维度保持一致，除此之外，如果不使用零填充，那么数据体的尺寸就会略微减少，在不断进行卷积的过程中，图像的边缘信息会过快地损失掉。

上面介绍了卷积神经网络中最重要的三种层结构：卷积层、池化层和全连接层，下面会介绍每一种层结构在 PyTorch 中是如何实现的。

4.3 PyTorch 卷积模块

PyTorch 作为一个深度学习库，卷积神经网络是其中一个最为基础的模块，卷积神经网络中所有的层结构都可以通过 nn 这个包调用，下面具体介绍如何调用每种层结构，以及每个函数中的参数。

4.3.1 卷积层

`nn.Conv2d()` 就是 PyTorch 中的卷积模块了，里面常用的参数有 5 个，分别是 `in_chan nels`，`out_channels`，`kernel_size`，`stride`，`padding`，除此之外还有参数 `dilation`，`groups`，`bias`。下面来解释每个参数的含义。

`in_channels` 对应的是输入数据体的深度；`out_channels` 表示输出数据体的深度；`kernel_size` 表示滤波器（卷积核）的大小，可以使用一个数字来表示高和宽相同的卷积核，比如 `kernel_size=3`，也可以使用不同的数字来表示高和宽不同的卷积核，比如 `kernel_size=(3, 2)`；`stride` 表示滑动的步长；`padding=0` 表示四周不进行零填充，而 `padding=1` 表示四周进行 1 个像素点的零填充；`bias` 是一个布尔值，默认 `bias=True`，表示使用偏置；`groups` 表示输出数据体深度上和输入数据体深度上的联系，默认 `groups=1`，也就是所有的输出和输入都是相关联的，如果 `groups=2`，这表示输入的深度被分割成两份，输出的深度也被分割成两份，它们之间分别对应起来，

所以要求输出和输入都必须要能被 groups 整除；dilation 表示卷积对于输入数据体的空间间隔，默认 dilation=1，可以直观地用图4.11表示。

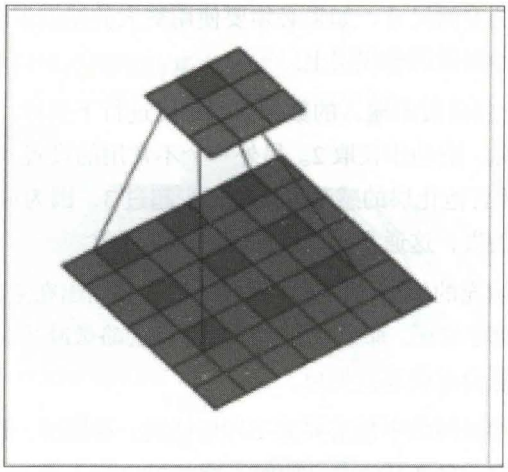


图 4.11　数据体的空间间隔

4.3.2 池化层

nn.MaxPool2d() 表示网络中的最大值池化，其中的参数有 kernel_size、stride、padding、dilation、return_indices、ceil_mode，下面解释一下它们各自的含义。

- kernel_size，stride，padding，dilation 之前卷积层已经介绍过了，是相同的含义；
- return_indices 表示是否返回最大值所处的下标，默认 return_indices=False；
- ceil_mode 表示使用一些方格代替层结构，默认 ceil_mode=False，一般都不会设置这些参数。
- nn.AvgPool2d() 表示均值池化，里面的参数和 nn.MaxPool2d() 类似，但多一个参数 count_incl ude_pad，这个参数表示计算均值的时候是否包含零填充，默认 count_include_pad=True。

一般使用较多的就是 nn.MaxPool2d() 和 nn.AvgPool2d()，另外 PyTorch 还提供了一些别的池化层，如 nn.LPPool2d()、nn.AdaptiveMaxPool2d() 等，这些运用较少，感兴趣的读者可以去查看官方文档。

全连接层和激活函数上一章已经介绍过，接下来构造一个简单的多层卷积神经网络。

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()  # b, 3, 32, 32
        layer1 = nn.Sequential()
        layer1.add_module('conv1', nn.Conv2d(3, 32, 3, 1, padding=1))
        # b, 32, 32, 32
        layer1.add_module('relu1', nn.ReLU(True))
        layer1.add_module('pool1', nn.MaxPool2d(2, 2))  # b, 32, 16, 16
        self.layer1 = layer1

        layer2 = nn.Sequential()
        layer2.add_module('conv2', nn.Conv2d(32, 64, 3, 1, padding=1))
        # b, 64, 16, 16
        layer2.add_module('relu2', nn.ReLU(True))
        layer2.add_module('pool2', nn.MaxPool2d(2, 2))  # b, 64, 8, 8
        self.layer2 = layer2

        layer3 = nn.Sequential()
        layer3.add_module('conv3', nn.Conv2d(64, 128, 3, 1, padding=1))
        #b, 128, 8, 8
        layer3.add_module('relu3', nn.ReLU(True))
        layer3.add_module('pool3', nn.MaxPool2d(2, 2))  #b, 128, 4, 4
        self.layer3 = layer3

        layer4 = nn.Sequential()
        layer4.add_module('fc1', nn.Linear(2048, 512))
        layer4.add_module('fc_relu1', nn.ReLU(True))
        layer4.add_module('fc2', nn.Linear(512, 64))
        layer4.add_module('fc_relu2', nn.ReLU(True))
        layer4.add_module('fc3', nn.Linear(64, 10))
        self.layer4 = layer4

    def forward(self, x):
        conv1 = self.layer1(x)
        conv2 = self.layer2(conv1)
        conv3 = self.layer3(conv2)
        fc_input = conv3.view(conv3.size(0), -1)
```

```
        fc_out = self.layer4(fc_input)
        return fc_out
model = SimpleCNN()
```

在上面的定义中，我们将卷积层、激活层和池化层组合在一起构成了一个层结构，定义了 3 个这样的层结构，最后定义了全连接层，输出 10。强烈建议在建立卷积层和池化层时通过参数计算一下输出的数据体大小，然后在代码旁边写出注释，这样在定义很复杂的网络结构时就不容易出错。

同时可以在 `forward` 中 `return` 添加中间层的输出结果，这样能很方便地得到网络的中间层输出。

还可以通过 `print(model)` 显示网络中定义了哪些层结构。

从图 4.12中可以看到这些层结构正如之前定义的一样，括号里面表示它的名字。下面会介绍如何提取网络中指定的层结构、参数，以及如何对参数进行自定义的初始化。

```
SimpleCNN (
  (layer1): Sequential (
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu1): ReLU (inplace)
    (pool1): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (layer2): Sequential (
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu2): ReLU (inplace)
    (pool2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (layer3): Sequential (
    (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu3): ReLU (inplace)
    (pool3): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (layer4): Sequential (
    (fc1): Linear (2048 -> 512)
    (fc_relu1): ReLU (inplace)
    (fc2): Linear (512 -> 64)
    (fc_relu2): ReLU (inplace)
    (fc3): Linear (64 -> 10)
  )
)
```

图 4.12　层结构

4.3.3　提取层结构

对于一个给定的模型，如果不想要模型中所有的层结构，只希望能够提取网络中的某一层或者几层，应该如何来实现呢?

首先看看 `nn.Module` 的几个重要属性。第一个是 `children()`，这个会返回下一级模块的迭代器，比如上面这个模型，它只会返回在 `self.layer1`, `self.layer2`, `self.layer3`，以及 `self.layer4` 上的迭代器，不会返回它们内部的东西；`modules()` 会返回模型中所有模块的迭代器，这样就有了一个好处，即它能够访问到最内层，比如

self.layer1.conv1 这个模块；还有一个与它们相对应的是 named_children() 属性以及 named_modules()，这两个不仅会返回模块的迭代器，还会返回网络层的名字。

下面来提取网络中我们需要的层，如果希望能够提取出前面两层，那么可以通过下面的办法来实现，得到如图4.13所示的结果。

```
new_model = nn.Sequential(*list(model.children())[:2])
```

```
Sequential (
  (0): Sequential (
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu1): ReLU (inplace)
    (pool1): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (1): Sequential (
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu2): ReLU (inplace)
    (pool2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
)
```

图 4.13 提取层的结果

如果希望提取出模型中所有的卷积层，可以像下面这样操作：

```
for layer in model.named_modules():
    if isinstance(layer[1], nn.Conv2d):
        conv_model.add_module(layer[0], layer[1])
```

使用 isinstance 可以判断这个模块是不是所需要的类型实例，这样就提取出了所有的卷积模块，得到如图4.14所示的结果。

```
Sequential (
  (layer1.conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (layer2.conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (layer3.conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

图 4.14 提取所有卷积模块

4.3.4 如何提取参数及自定义初始化

有时候提取出的层结构并不够，还需要对里面的参数进行初始化，那么如何提取出网络的参数并对其初始化呢?

首先 nn.Module 里面有两个特别重要的关于参数的属性，分别是 named_parameters() 和 parameters()。named_para meters() 是给出网络层的名字和参数的迭代器，parameter s() 会给出一个网络的全部参数的迭代器。

```
for param in model.named_parameters():
```

```
    print(param[0])
```

可以得到每一层参数的名字，如图 4.15所示。

```
layer1.conv1.weight
layer1.conv1.bias
layer2.conv2.weight
layer2.conv2.bias
layer3.conv3.weight
layer3.conv3.bias
layer4.fc1.weight
layer4.fc1.bias
layer4.fc2.weight
layer4.fc2.bias
layer4.fc3.weight
layer4.fc3.bias
```

图 4.15　得到每一层参数的名字

如何对权重做初始化呢？非常简单，因为权重是一个 Variable，所以只需要取出其中的 data 属性，然后对它进行所需要的处理就可以了。

```
for m in model.modules():
    if isinstance(m, nn.Conv2d):
        init.normal(m.weight.data)
        init.xavier_normal(m.weight.data)
        init.kaiming_normal(m.weight.data)
        m.bias.data.fill_(0)
    elif isinstance(m, nn.Linear):
        m.weight.data.normal_()
```

通过上面的操作，对将卷积层中使用 PyTorch 里面提供的方法的权重进行初始化，这样就能够使用任意我们想使用的初始化，甚至我们可以自己定义初始化方法并对权重进行初始化。

4.4　卷积神经网络案例分析

上面部分介绍了 PyTorch 中的卷积模块，接下来将会介绍几个卷积神经网络的案例，通过案例入手来介绍卷积神经网络的结构设计。

4.4.1 LeNet

LeNet 是整个卷积神经网络的开山之作，1998 年由 LeCun 提出，它的结构特别简单，我们能够由此入手，一步一步地进入时下最为流行的卷积神经网络结构。

首先，LeNet 的网络结构如图4.16所示。

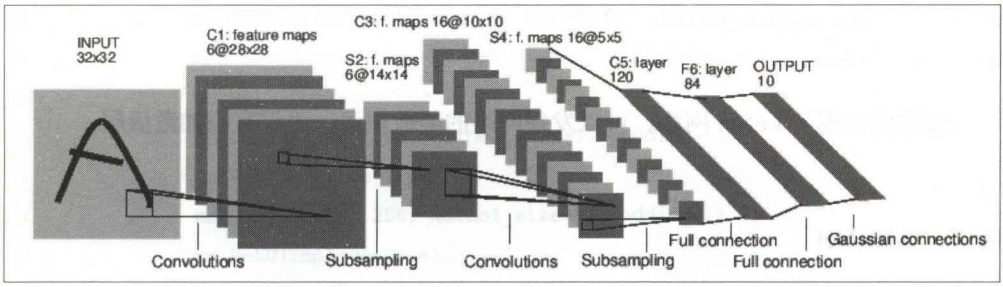


图 4.16 LeNet 网络结构

从图 4.16 可以看出整个网络结构特别清晰，一共有 7 层，其中 2 层卷积和 2 层池化层交替出现，最后输出 3 层全连接层得到整体的结果。

针对这个简单的网络，可以自己操作。

```
class Lenet(nn.Module):
    def __init__(self):
        super(Lenet, self).__init__()

        layer1 = nn.Sequential()
        layer1.add_module('conv1', nn.Conv2d(1, 6, 3, padding=1))
        layer1.add_module('pool1', nn.MaxPool2d(2, 2))
        self.layer1 = layer1

        layer2 = nn.Sequential()
        layer2.add_module('conv2', nn.Conv2d(6, 16, 5))
        layer2.add_module('poo2', nn.MaxPool2d(2, 2))
        self.layer2 = layer2

        layer3 = nn.Sequential()
        layer3.add_module('fc1', nn.Linear(400, 120))
        layer3.add_module('fc2', nn.Linear(120, 84))
        layer3.add_module('fc3', nn.Linear(84, 10))
```

```
        self.layer3 = layer3

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.layer3(x)
        return x
```

这样就实现了 LeNet 网络，可以发现网络的层数很浅，也没有添加激活层。

4.4.2 AlexNet

接下来要介绍 2012 年在 ImageNet 竞赛上面大放异彩的 AlexNet，它以领先第二名 10 % 的准确率夺得冠军，并且成功地向世界展示了深度学习的威力。

首先看看 AlexNet 的网络结构，如图 4.17所示。

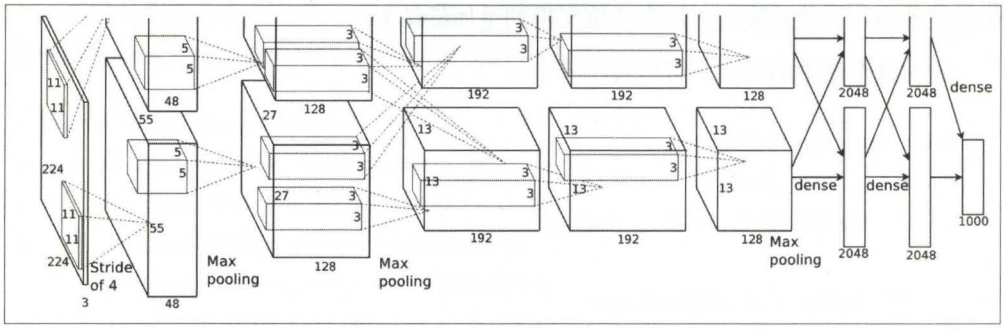


图 4.17　AlexNet 网络结构

图4.17可能让你看得眼花缭乱，其实这是因为当时 GPU 计算能力不强，而 AlexNet 又比较复杂，所以 Alex 使用了两个 GPU 并行来做运算，现在已经完全可以用一个 GPU 代替了。

AlexNet 网络相对于 LeNet，层数更深，同时第一次引入了激活层 ReLU，在全连接层引入了 Dropout 层防止过拟合。

实现 AlexNet 的网络结构如下。

```
class AlexNet(nn.Module):
    def __init__(self, num_classes):
        super(AlexNet, self).__init__()
```

```
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2), )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes), )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```

这是 ImageNet 竞赛史上第一次基于卷积神经网络的模型得到冠军，从此掀起了深度学习在计算机视觉上的革命。

4.4.3 VGGNet

VGGNet 是 ImageNet 2014 年的亚军，总结起来就是它使用了更小的滤波器，同时使用了更深的结构，AlexNet 只有 8 层网络，而 VGGNet 有 16 层～19 层网络，也不像

AlexNet 使用 11×11 那么大的滤波器，它只使用 3×3 的卷积滤波器和 2×2 的大池化层。图4.18是 AlexNet 和 VGGNet 的对比图。

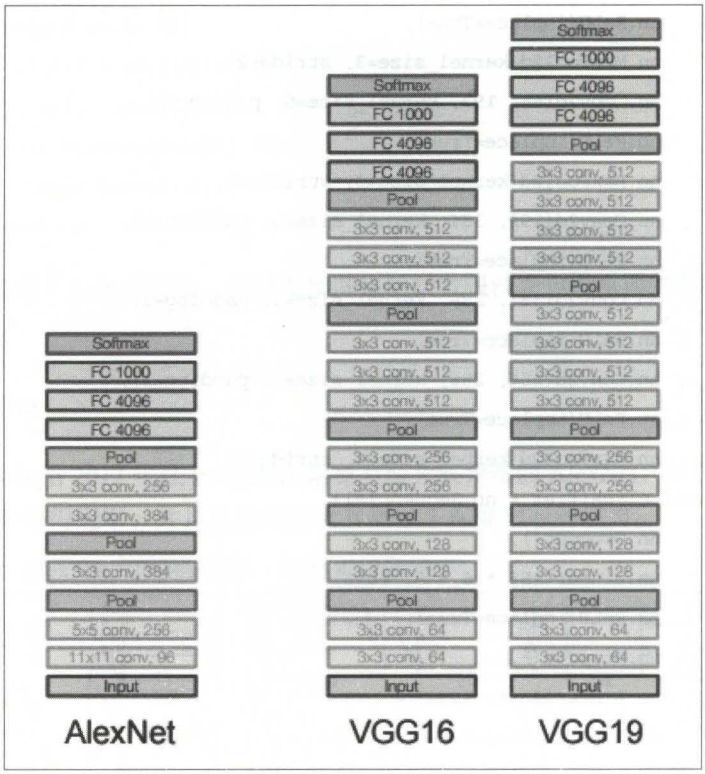


图 4.18　AlexNet 和 VGGNet 对比图

它之所以使用很多小的滤波器，是因为层叠很多小的滤波器的感受野和一个大的滤波器的感受野是相同的，还能减少参数，同时有更深的网络结构。

同样可以实现如下。

```
class VGG(nn.Module):
    def __init__(self, num_classes):
        super(VGG, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
```

```
            nn.ReLU(True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size=2, stride=2), )
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes), )
        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
```

```
48          x = x.view(x.size(0), -1)
49          x = self.classifier(x)
```

其实可以看出 VGG 只是对网络层进行不断的堆叠，并没有进行太多的创新，而增加深度确实可以一定程度改善模型效果。

4.4.4 GoogLeNet

GoogLeNet 也叫 InceptionNet，是在 2014 年被提出的，如今已经进化到了 v4 版本，下面介绍它最核心的部分。

GoogLeNet 采取了比 VGGNet 更深的网络结构，一共有 22 层，但是它的参数却比 AlexNet 少了 12 倍，同时有很高的计算效率，因为它采用了一种很有效的 Inception 模块，而且它也没有全连接层，是 2014 年比赛的冠军。

先看看 GoogLeNet 的网络结构和其中最为创新的 Inception 模块，如图4.19所示。

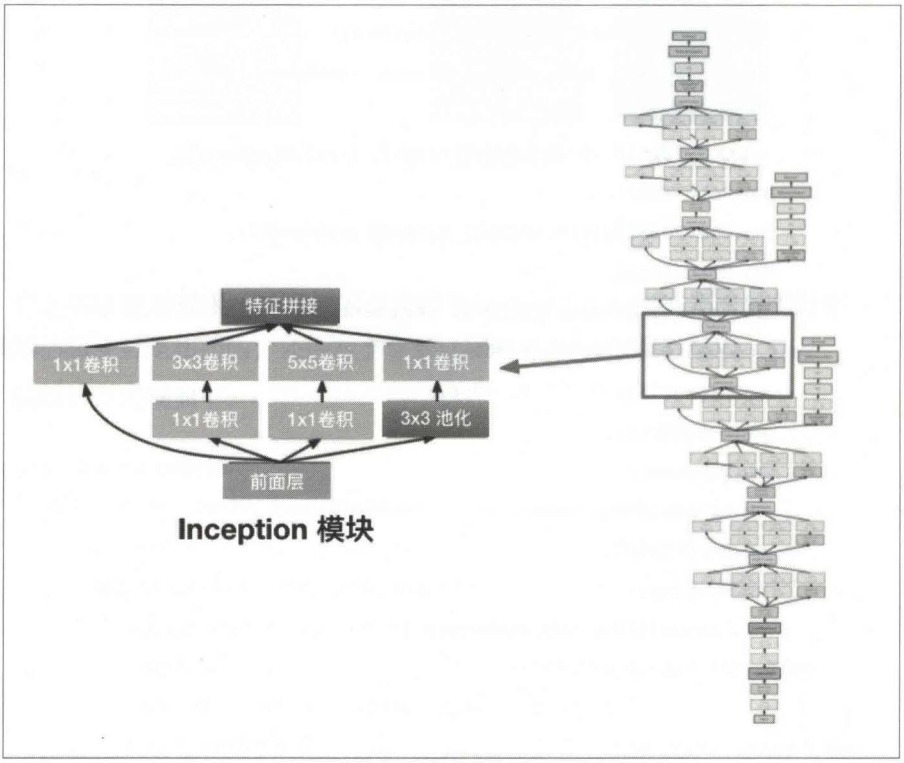


图 4.19 Inception 模块

Inception 模块设计了一个局部的网络拓扑结构，然后将这些模块堆叠在一起形成一个抽象层网络结构。具体来说就是运用几个并行的滤波器对输入进行卷积和池化，这些滤波器有不同的感受野，最后将输出的结果按深度拼接在一起形成输出层。

这样的网络结构非常新颖，正是由于这种网络结构，GoogLeNet 才能够取得如此大的成功，但是这种网络结构有一个小问题，就是参数太多，导致计算复杂。

为了解决这个问题，GoogLeNet 又推出了下一个版本，这个版本对 Inception 模块有了新的设计，如图4.20所示。

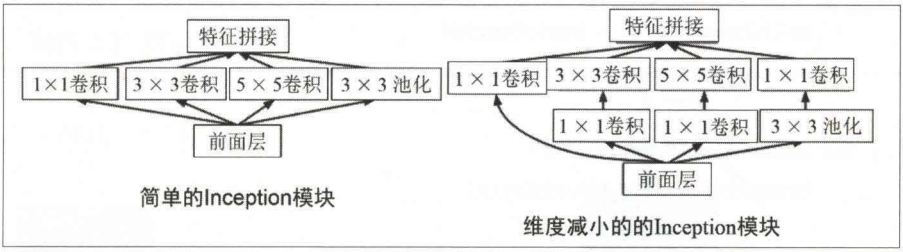


图 4.20 Inception 模块的新设计

这个模块增加了一些 1×1 的卷积层来降低输入层的维度，使网络参数减少，从而减少了网络的复杂性。

下面实现 GoogLeNet 中的 Inception 模块，整个 GoogLeNet 都是由这些 Inception 模块组成的。

```
class BasicConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, **kwargs):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **
        kwargs)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)

class Inception(nn.Module):
    def __init__(self, in_channels, pool_features):
        super(Inception, self).__init__()
        self.branch1x1 = BasicConv2d(in_channels, 64, kernel_size=1)
```

```
16
17          self.branch5x5_1 = BasicConv2d(in_channels, 48, kernel_size=1)
18          self.branch5x5_2 = BasicConv2d(48, 64, kernel_size=5, padding=2)
19
20          self.branch3x3dbl_1 = BasicConv2d(in_channels, 64, kernel_size=1)
21          self.branch3x3dbl_2 = BasicConv2d(64, 96, kernel_size=3, padding=1)
22          self.branch3x3dbl_3 = BasicConv2d(96, 96, kernel_size=3, padding=1)
23
24          self.branch_pool = BasicConv2d(
25              in_channels, pool_features, kernel_size=1)
26
27      def forward(self, x):
28          branch1x1 = self.branch1x1(x)
29
30          branch5x5 = self.branch5x5_1(x)
31          branch5x5 = self.branch5x5_2(branch5x5)
32
33          branch3x3dbl = self.branch3x3dbl_1(x)
34          branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
35          branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)
36
37          branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
38          branch_pool = self.branch_pool(branch_pool)
39
40          outputs = [branch1x1, branch5x5, branch3x3dbl, branch_pool]
41          return torch.cat(outputs, 1)
```

首先定义一个最基础的卷积模块，然后根据这个模块定义了 1×1，3×3 和 5×5 的模块和一个池化层，最后使用 `torch.cat()` 将它们按深度拼接起来，得到输出结果。

4.4.5 ResNet

ResNet 是 2015 年 ImageNet 竞赛的冠军，由微软研究院提出，通过残差模块能够成功地训练高达 152 层深的神经网络。

ResNet 最初的设计灵感来自这个问题：在不断加深神经网络的时候，会出现一个 Degradation，即准确率会先上升然后达到饱和，再持续增加深度则会导致模型准确率

下降。

这并不是过拟合的问题，因为不仅在验证集上误差增加，训练集本身误差也会增加。假设一个比较浅的网络达到了饱和的准确率，那么在后面加上几个恒等映射层，误差不会增加，也就说更深的模型起码不会使得模型效果下降。

这里提到的使用恒等映射直接将前一层输出传到后面的思想，就是 ResNet 的灵感来源。假设某个神经网络的输入是 x，期望输出是 $H(x)$，如果直接把输入 x 传到输出作为初始结果，那么此时需要学习的目标就是 $F(x) = H(x) - x$，也就是下面这个残差模块，如图 4.21 所示。

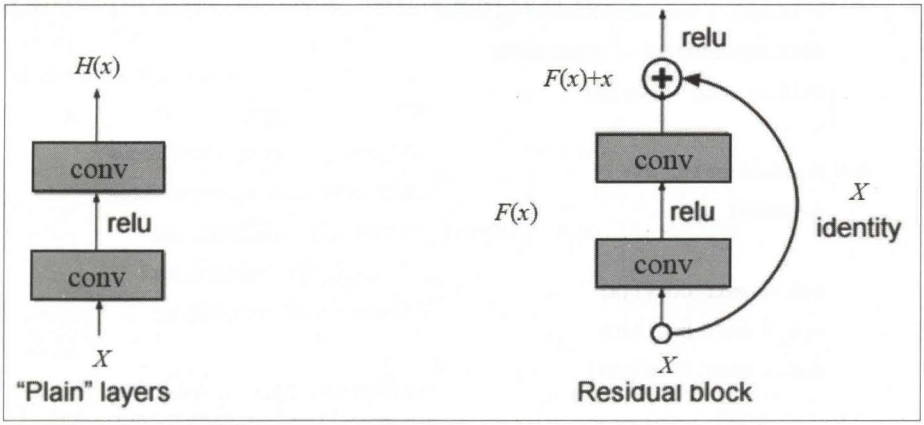


图 4.21 普通网络与 ResNET 的残差学习单元

图 4.21左边是一个普通的网络，右边是一个 ResNet 的残差学习单元，ResNet 相当于将学习目标改变了，不再是学习一个完整的输出 $H(x)$，而是学习输出和输入的差别 $H(x) - x$，即残差。

ResNet 因为残差模块的存在，使整个网络可以训练高达 152 层，下面只用 ResNet 网络中的残差模块举例。

```
def conv3x3(in_planes, out_planes, stride=1):
    "3x3 convolution with padding"
    return nn.Conv2d(
        in_planes,
        out_planes,
        kernel_size=3,
        stride=stride,
        padding=1,
        bias=False)
```

```
10
11
12 class BasicBlock(nn.Module):
13     def __init__(self, inplanes, planes, stride=1, downsample=None):
14         super(BasicBlock, self).__init__()
15         self.conv1 = conv3x3(inplanes, planes, stride)
16         self.bn1 = nn.BatchNorm2d(planes)
17         self.relu = nn.ReLU(inplace=True)
18         self.conv2 = conv3x3(planes, planes)
19         self.bn2 = nn.BatchNorm2d(planes)
20         self.downsample = downsample
21         self.stride = stride
22
23     def forward(self, x):
24         residual = x
25
26         out = self.conv1(x)
27         out = self.bn1(out)
28         out = self.relu(out)
29
30         out = self.conv2(out)
31         out = self.bn2(out)
32
33         if self.downsample is not None:
34             residual = self.downsample(x)
35
36         out += residual
37         out = self.relu(out)
38
39         return out
```

从 `forward` 的最后一行，能够看出网络将最开始的 x 加到了输出当中，形成了残差结构。

除了这些比较出名的网络之外，卷积神经网络的世界中还有很多别的网络，比如 Network in Network、Highway Network 等，大家可以自己去看相关的论文。另外并不需要重复造轮子，PyTorch 中早就为我们实现了上面介绍过的这些网络，都在 `torchvision.model` 里面，同时大部分网络都有预训练好的参数，这些预训练好的网络为后面介绍的

迁移学习和微调做了很好的铺垫。

4.5 再实现 MNIST 手写数字分类

至今为止，已经介绍了很多很高级的网络结构，是不是跃跃欲试了呢？记得上一节我们使用简单的多层全连接神经网络在 MNIST 数据上训练，最后验证集达到了 98% 的准确率，能不能使用卷积神经网络来进一步提升网络的准确率呢？答案是肯定的，下面就来写一个多层卷积神经网络，使它在 MNIST 数据集上达到 99% 的验证集准确率。

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3),  # b, 16, 26, 26
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True))

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3),  # b, 32, 24, 24
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)  #b, 32, 12, 12
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3),  #b, 64, 10, 10
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True))

        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3),  #b, 128, 8, 8
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)  #b, 128, 4, 4
        )

        self.fc = nn.Sequential(
```

```
            nn.Linear(128 * 4 * 4, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 10))

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

上面这个简单的卷积神经网络是运用之前所学到的知识来建立的，里面有 4 层卷积，2 层最大池化，卷积之后使用批标准化加快收敛速度，使用 ReLU 激活函数增加非线性，最后使用全连接层输出分类得分。最后再测试一下网络的结果，可以看到图4.22所示的测试集准确率已经达到了 99.31%，比之前使用的 3 层全连接神经网络要高。可以看到通过增加网络的深度和复杂化网络的结构提高网络的准确率是可行的，下面我们将从数据方面出发来提高网络的准确率。

```
Epoch [16/20], Iter [100/468] Loss: 0.0349, Acc: 0.9922
Epoch [16/20], Iter [200/468] Loss: 0.0004, Acc: 1.0000
Epoch [16/20], Iter [300/468] Loss: 0.0000, Acc: 1.0000
Epoch [16/20], Iter [400/468] Loss: 0.0003, Acc: 1.0000
Epoch [17/20], Iter [100/468] Loss: 0.0091, Acc: 0.9922
Epoch [17/20], Iter [200/468] Loss: 0.0051, Acc: 1.0000
Epoch [17/20], Iter [300/468] Loss: 0.0034, Acc: 1.0000
Epoch [17/20], Iter [400/468] Loss: 0.0030, Acc: 1.0000
Epoch [18/20], Iter [100/468] Loss: 0.0001, Acc: 1.0000
Epoch [18/20], Iter [200/468] Loss: 0.0109, Acc: 0.9922
Epoch [18/20], Iter [300/468] Loss: 0.0002, Acc: 1.0000
Epoch [18/20], Iter [400/468] Loss: 0.0000, Acc: 1.0000
Epoch [19/20], Iter [100/468] Loss: 0.0153, Acc: 0.9922
Epoch [19/20], Iter [200/468] Loss: 0.0013, Acc: 1.0000
Epoch [19/20], Iter [300/468] Loss: 0.0006, Acc: 1.0000
Epoch [19/20], Iter [400/468] Loss: 0.0032, Acc: 1.0000
Epoch [20/20], Iter [100/468] Loss: 0.0002, Acc: 1.0000
Epoch [20/20], Iter [200/468] Loss: 0.0095, Acc: 0.9922
Epoch [20/20], Iter [300/468] Loss: 0.0001, Acc: 1.0000
Epoch [20/20], Iter [400/468] Loss: 0.0084, Acc: 0.9922
Test Accuracy of the model on the 10000 test images: 0.9931
```

图 4.22　测试集准确率

4.6 图像增强的方法

前面的部分专注于卷积神经网络的层结构介绍，同时还介绍了到目前为止比较出名的卷积神经网络，接着使用比较复杂的卷积神经网络提高了 MNIST 数据集的准确率。下面将从另外的角度——图像增强的方面入手，提高模型的准确率和泛化能力。

一直以来，图像识别这一计算机视觉的核心问题都面临很多挑战，同一个物体在不同情况下都会得出不同的结论。

对于一张照片，我们看到的是一些物体，而对于计算机而言，它看到的是一些像素点，如图 4.23所示。

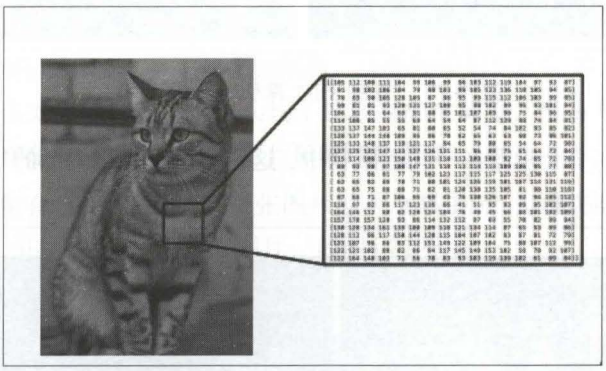


图 4.23 计算机识别的图片

如果拍摄照片的照相机位置发生了改变，那么拍得的图片对于我们而言，变化很小，但是对于计算机而言，图片的像素变化是很大的。拍照时的光照条件也是很重要的一个影响因素：光照太弱，照片里的物体会和背景融为一体，它们的像素点就会很接近，计算机就无法正确识别出物体，如图4.24所示。

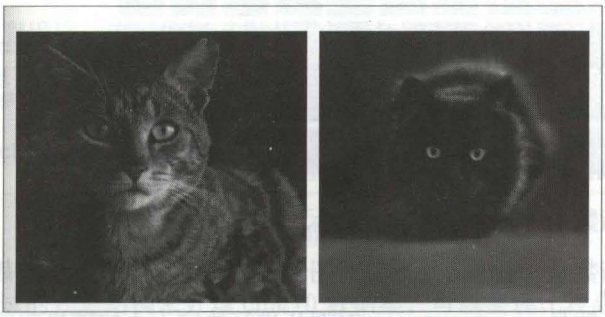


图 4.24 光照对计算机识别的影响

除此之外，物体本身的变形也会对计算机识别造成障碍，比如一只猫是趴着的，计

算机能够识别它，但如果猫换了一个姿势，变成躺着的状态，那么计算机就无法识别了，如图 4.25所示。

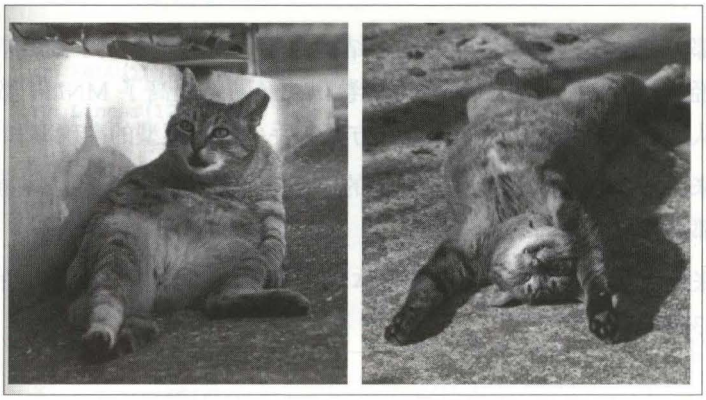


图 4.25　姿势对计算机识别的影响

最后，物体本身会隐藏在一些遮蔽物中，这样物体只呈现局部的信息，计算机也难以识别，如图4.26所示。

图 4.26　是否有遮蔽物对计算机识别的影响

针对这些问题，我们希望能够对原始图片进行增强，在一定程度上解决部分问题。在 PyTorch 中已经内置了一些图像增强的方法，不需要再繁琐地去实现，只需要简单的调用。

`torchvision.transforms` 包括所有图像增强的方法。

- 第一个函数是 `Scale`，对图片的尺度进行缩小和放大；
- 第二个函数是 `CenterCrop`，对图像正中心进行给定大小的裁剪；
- 第三个函数是 `RandomCrop`，对图片进行给定大小的随机裁剪；
- 第四个函数是 `RandomHorizaontalFlip`，对图片进行概率为 0.5 的随机水平翻转；

- 第五个函数是 `RandomSizedCrop`，首先对图片进行随机尺寸的裁剪，然后对裁剪的图片进行一个随机比例的缩放，最后将图片变成给定的大小，这在 Inception Net 中比较流行；
- 最后一个是 `Pad`，对图片进行边界零填充。

上面介绍了 PyTorch 内置的一些图像增强的方法，还有更多的增强方法，可以使用 OpenCV 或者 PIL 等第三方图形库实现。在网络的训练中图像增强是一种常见、默认的做法，对多任务进行图像增强之后都能够在一定程度上提升任务的准确率。

4.7 实现 cifar10 分类

前面我们处理过 MNIST 手写数字数据集，这些数据集比较简单，大小是 28×28 的灰度图，没有什么太多的特征，虽然实现了比较高的准确率，但是并没有太大的说服力，我们希望能够在更加复杂的数据集上有更好的表现。

cifar10 数据集有 60000 张图片，每张图片的大小都是 32×32 的三通道的彩色图，一共是 10 种类别，每种类别有 6000 张图片，如图4.27所示。

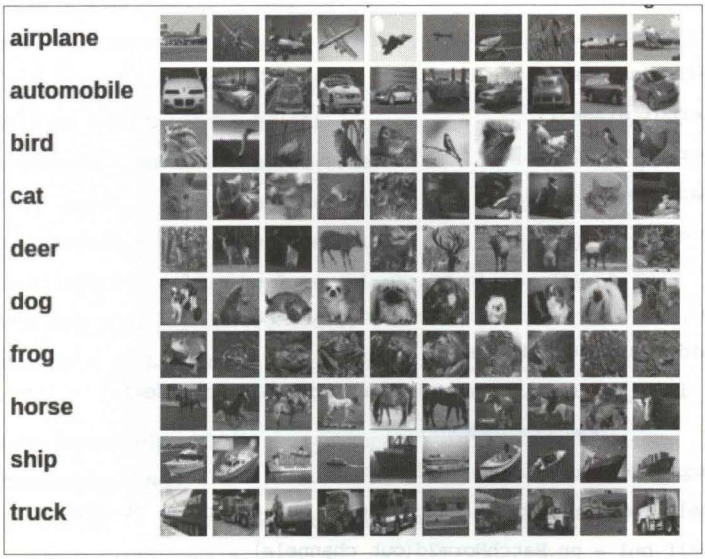


图 4.27 cifar 数据集

使用前面讲过的残差结构来处理 cifar10 数据集，可以实现比较高的准确率。

首先进行图像增强，使用前面介绍的增强方式。

```
train_transform = transforms.Compose([
```

```
    transforms.Scale(40),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
```

注意只对训练图片进行图像增强，提高其泛化能力，对于测试集，仅对其中心化，不做其他的图像增强。

下面先定义好 resnet 的基本模块。

```
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(
        in_channels,
        out_channels,
        kernel_size=3,
        stride=stride,
        padding=1,
        bias=False)

# Residual Block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1,
                 downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
```

```
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

和前面介绍的内容一样，先定义残差模块，再将残差模块拼接起来，注意其中的维度变化。

```
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[0], 2)
        self.layer3 = self.make_layer(block, 64, layers[1], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(
            block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
```

```
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

最后在 cifar10 的数据集上跑 50 个 epoch，实现 93.6 % 的训练集准确率，86.23% 的验证集准确率，因为这里只跑了 50 次，所以还有一定的提升空间。同时使用更深的残差和更多的训练技巧能实现更好的实验结果，如图4.28所示。

```
Epoch [44/50], Iter [300/391] Loss: 0.196086 Acc: 0.931380
Epoch [45/50], Iter [100/391] Loss: 0.189617 Acc: 0.936875
Epoch [45/50], Iter [200/391] Loss: 0.190259 Acc: 0.934531
Epoch [45/50], Iter [300/391] Loss: 0.193326 Acc: 0.933307
Epoch [46/50], Iter [100/391] Loss: 0.188115 Acc: 0.933047
Epoch [46/50], Iter [200/391] Loss: 0.187997 Acc: 0.933867
Epoch [46/50], Iter [300/391] Loss: 0.189580 Acc: 0.933307
Epoch [47/50], Iter [100/391] Loss: 0.196518 Acc: 0.931641
Epoch [47/50], Iter [200/391] Loss: 0.192699 Acc: 0.931758
Epoch [47/50], Iter [300/391] Loss: 0.191035 Acc: 0.932760
Epoch [48/50], Iter [100/391] Loss: 0.185273 Acc: 0.935703
Epoch [48/50], Iter [200/391] Loss: 0.187424 Acc: 0.933984
Epoch [48/50], Iter [300/391] Loss: 0.190616 Acc: 0.933568
Epoch [49/50], Iter [100/391] Loss: 0.183902 Acc: 0.936172
Epoch [49/50], Iter [200/391] Loss: 0.189872 Acc: 0.933477
Epoch [49/50], Iter [300/391] Loss: 0.189366 Acc: 0.933880
Epoch [50/50], Iter [100/391] Loss: 0.191345 Acc: 0.934219
Epoch [50/50], Iter [200/391] Loss: 0.185211 Acc: 0.935430
Epoch [50/50], Iter [300/391] Loss: 0.184794 Acc: 0.936120
Accuracy of the model on the test images: 86.23 %%
```

图 4.28　实验结果

上面的所有小节中，从计算机视觉的任务入手，引入了卷积神经网络，介绍了卷积神经网络的基本层结构，接着介绍了时下非常出名的卷积神经网络案例，最后引入图像增强的技巧，使用 PyTorch 进行网络搭建和训练。

下一章将介绍一种非常适合处理序列数据和自然语言问题的神经网络——循环神经网络。

第 5 章

循环神经网络

对于人类而言，以前见过的事物会在脑海里面留下记忆，虽然随后记忆会慢慢消失，但是每当经过提醒，人们往往能够重拾记忆。在神经网络的研究中，让模型充满记忆力的研究很早便开始了，Saratha Sathasivam 于 1982 年提出了霍普菲尔德网络，但是由于它实现困难，在提出的时候也没有很好的应用场景，所以逐渐被遗忘。深度学习的兴起又让人们重新开始研究循环神经网络（Recurrent Neural Network），并在序列问题和自然语言处理等领域取得很大的成功。

这一章将从循环神经网络的基本结构出发，介绍循环神经网络的基本模型和存在的问题，接着介绍循环神经网络目前使用最多的两种变式：LSTM 和 GRU，然后阐述一个使用 PyTorch 处理时序问题的实例，接着介绍循环神经网络在自然语言处理中的应用及其 PyTorch 实现，最后介绍它在机器翻译、语音识别等领域中的应用。

5.1 循环神经网络

前一章介绍了卷积神经网络，卷积神经网络相当于人类的视觉，但是它并没有记忆能力，所以它只能处理一种特定的视觉任务，没办法根据以前的记忆来处理新的任务。那么记忆力对于网络而言到底是不是必要的呢？很显然在某些问题上是必要的，比如，在一场电影中推断下一个时间点的场景，这个时候仅依赖于现在的情景并不够，需要依赖于前面发生的情节。对于这样一些不仅依赖于当前情况，还依赖于过去情况的问题，传统的神经网络结构无法很好地处理，所以基于记忆的网络模型是必不可少的。

循环神经网络的提出便是基于记忆模型的想法，期望网络能够记住前面出现的特征，并依据特征推断后面的结果，而且整体的网络结构不断循环，因为得名循环神经网络。

5.1.1 问题介绍

前面介绍过多层全连接网络和卷积神经网络，可以发现这些网络不需要记忆的特性也能处理对应的任务，下面举一个具体的例子引出循环神经网络。

首先我们来看看下面这两句话，如图5.1与图5.2所示。

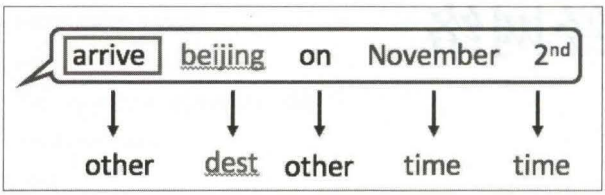


图 5.1 到达北京

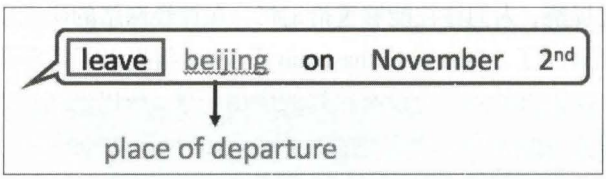


图 5.2 离开北京

第一句话表达到达北京的意思，第二句话表达离开北京的意思。对于这样一个任务，如果网络只输入“beijing”这个词而没有记忆的特性，那么对于这两句话，网络会输出相同的结果。但是如果网络能够记忆“beijing”前面的词，它就会预测出不同的结果，所以说神经网络需要记忆的特性。

5.1.2 循环神经网络的基本结构

循环神经网络的基本结构特别简单，就是将网络的输出保存在一个记忆单元中，这个记忆单元和下一次的输入一起进入神经网络中。使用一个简单的两层网络作为示范，在它的基础上扩充为循环神经网络的结构，我们用图 5.3简单地表示。

可以看到网络在输入的时候会联合记忆单元一起作为输入，网络不仅输出结果，还会将结果保存到记忆单元中，图 5.3 就是一个最简单的循环神经网络在一次输入时的结构示意图。

从上面的原理中可以发现，输入序列的顺序改变，会改变网络的输出结果，这是因为记忆单元的存在，使得两个序列在顺序改变之后记忆单元中的元素也改变了，所以会影响最终的输出结果。

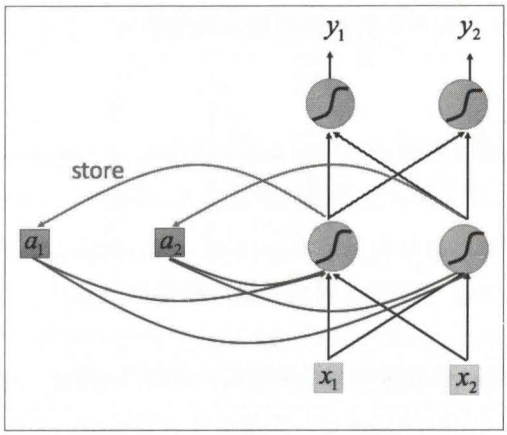


图 5.3 将一个数据点传入网络

图5.3是序列中一个数据点传入网络的示意图，那么整个序列如何传入网络呢？将序列中的每个数据点依次传入网络即可，如图5.4所示。

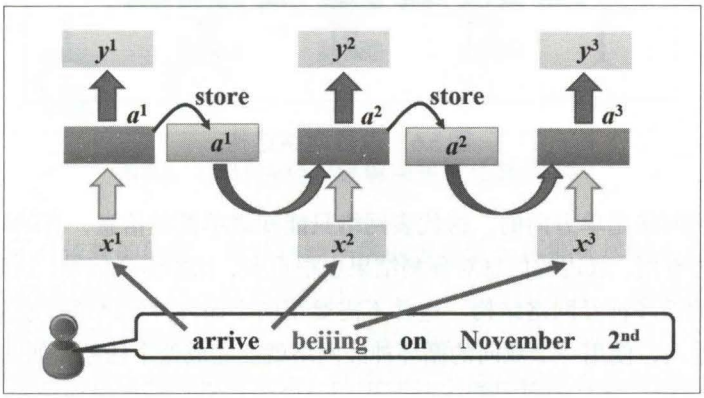


图 5.4 将整个序列传入网络

无论序列有多长，都能不断输入网络，最终得到结果。可能看到这里，读者会有一些疑问，图5.4中每一个网络是不是都是独立的权重？对于这个问题，先考虑一下如果是不同的序列，那么图 5.4 中格子的数目就是不同的，对于一个网络结构，不太可能出现这种参数数目变化的情况。

事实上，这里再次使用了参数共享的概念，也就是说虽然上面有三个格子，其实它们都是同一个格子，而网络的输出依赖于输入和记忆单元，可以用图 5.5表示。

如图5.5所示，左边就是循环神经网络实际的网络流，右边是将其展开的结果，可以看到网络中具有循环结构，这也是循环神经网络名字的由来。同时根据循环神经网络的结构也可以看出它在处理序列类型的数据上具有天然的优势，因为网络本身就是一个序列结构，这也是所有循环神经网络最本质的结构。

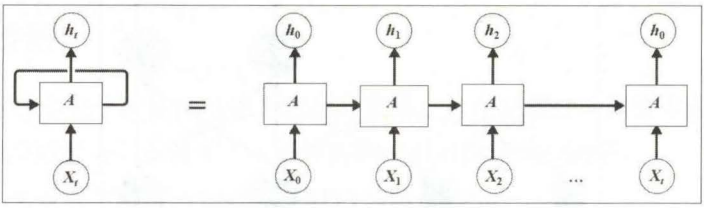


图 5.5　网络的输入和记忆单元

循环神经网络也可以有很深的网络层结构，如图 5.6所示。

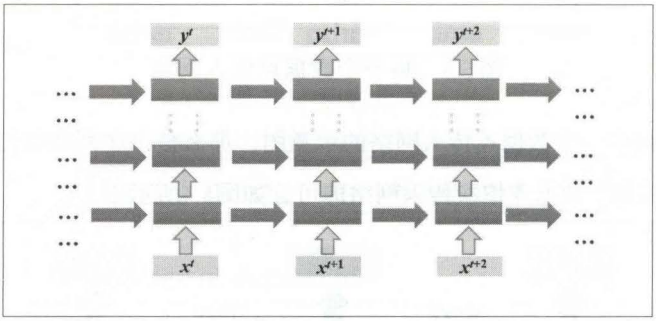


图 5.6　深层网络结构

可以看到网络是单方向的，这代表网络只能知道单侧的信息，有的时候序列的信息不只是单边有用，双边的信息对预测结果也很重要，比如语音信号，这时候就需要看到两侧信息的循环神经网络结构。这并不需要用两个循环神经网络分别从左右两边开始读取序列输入，使用一个双向的循环神经网络就能完成这个任务，如图 5.7所示。

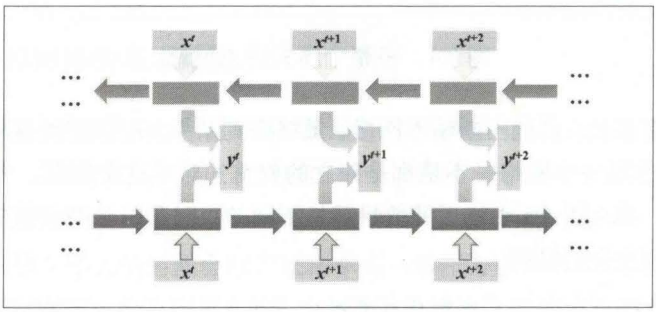


图 5.7　双向循环神经网络

使用双向循环神经网络，网络会先从序列的正方向读取数据，再从反方向读取数据，最后将网络输出的两种结果合在一起形成网络的最终输出结果。

5.1.3 存在的问题

根据前面介绍的内容可以了解到循环神经网络具有特别好的记忆特性，能够将记忆内容应用到当前情景下，但是随后人们发现网络的记忆能力并没有想象中那么有效。

记忆最大的问题在于它有遗忘性，我们总是更加清楚地记得最近发生的事情而遗忘很久之前发生的事情，循环神经网络同样有这样的问题。如果一项任务需要依赖近期的信息来预测结果，循环神经网络往往具有比较好的表现，比如给出下面一句话“我住在中国，我会讲中文”，使用循环神经网络就能够依据前面内容中的“中国”来预测后面的“中文”。循环神经网络能够很好地解决这种短时依赖的问题，如图 5.8所示。

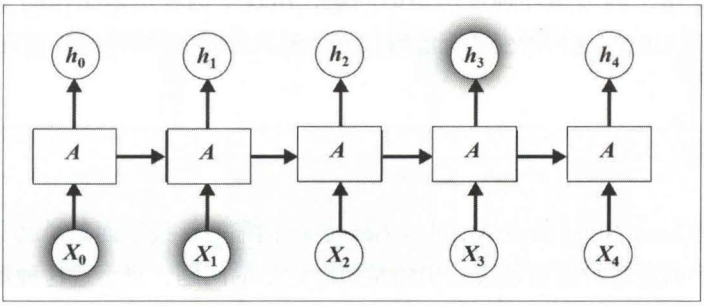


图 5.8 循环神经网络解决短时依赖问题

但是对于长时依赖的问题，循环神经网络的表现就不再那么尽如人意。比如将上一个问题中的这句话拆开，在一篇文章的开头写上“我住在中国”，希望循环神经网络在文章的末尾能够预测“我会讲中文”，这时网络就没办法很好地预测这个结果，因为记忆的信息和预测位置之间的跨度太大，网络往往不能记忆这么长时间的信息，而且随着时间跨度越来越大，循环神经网络也越来越难以学习这些信息，如图5.9所示。

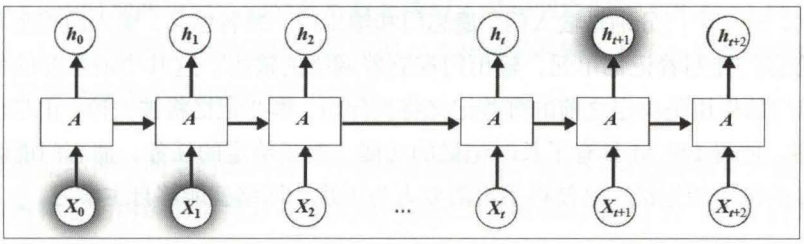


图 5.9 循环神经网络记忆长时间信息

这些问题导致循环神经网络在实际应用上一直没办法取得很好的效果，但是这些问题能不能解决呢？从理论上说，循环神经网络是完全能够解决这种长时依赖问题的，可以针对具体的问题，人为挑选一些特定的参数来解决，但这并不具有普适性，因为循环神经网络没有办法自己决定该挑选哪些参数。1994 年 Bengio 甚至发表论文论述长时依赖问题的原因以及为什么它难以解决。

早期循环神经网络的发展存在长时依赖问题，后来依据循环神经网络的基本结构，有人提出了一些特定的变式，这些变式解决了长时依赖的问题，并在此后循环神经网络的应用中成为主流，下面就来介绍目前循环神经网络最为流行的两种结构：LSTM 和 GRU。

5.2 循环神经网络的变式：LSTM 与 GRU

上一节介绍了因为循环神经网络的问题，出现了两种网络结构的变式：一种叫 LSTM，一种叫 GRU，这两种变式都能够很好地解决长时依赖的问题。首先介绍 LSTM。

5.2.1 LSTM

LSTM 是 Long Short Term Memory Networks 的缩写，按字面翻译就是长的短时记忆网络，从字面意思知道它解决的仍然是短时记忆的问题，只不过这种短时记忆比较长，能在一定程度上解决长时依赖的问题。LSTM 的网络结构是 1997 年由 Hochreiter 和 Schmidhuber 提出的，随后这种网络结构变得非常流行，有很多人跟进相关的工作解决了很多实际的问题，现在 LSTM 仍然被广泛地使用。

循环神经网络的结构都是链式循环的网络结构，LSTM 的网络结构大体上也是这样的结构，不过 LSTM 在网络的内部有着更加复杂的结构，所以它能够处理长时依赖的问题。

LSTM 的抽象网络结构示意图如图5.10所示。从图 5.10 中可以看出 LSTM 由三个门来控制，这三个门分别是输入门、遗忘门和输出门。顾名思义，输入门控制着网络的输入，遗忘门控制着记忆单元，输出门控制着网络的输出。这其中最重要的就是遗忘门，遗忘门的作用是决定之前的哪些记忆将被保留，哪些记忆将被去掉，正是由于遗忘门的作用，使得 LSTM 具有了长时记忆的功能，对于给定的任务，遗忘门能够自己学习保留多少以前的记忆，这使得不再需要人为干扰，网络就能够自主学习。

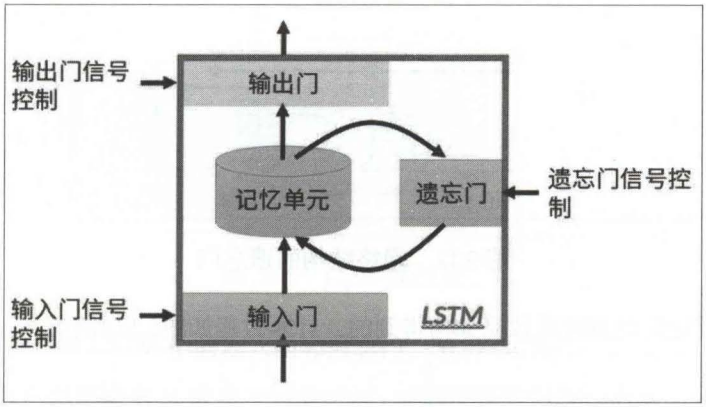


图 5.10 LSTM 的抽象网络结构

下面从具体的内部结构来解释 LSTM 的网络流过程，首先给出 LSTM 内部的结构示意图，如图 5.11所示。

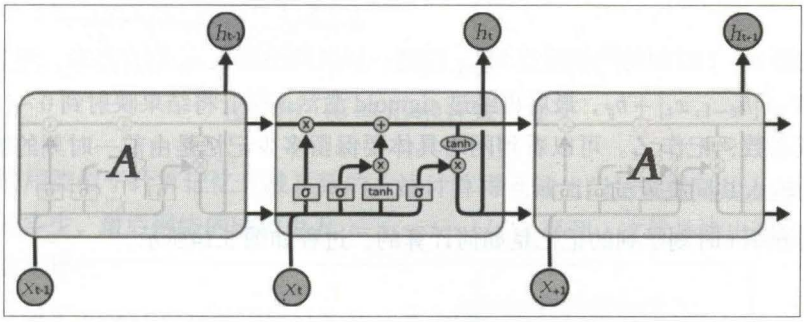


图 5.11 LSTM 内部结构

C_{t-1} 作为上一步 $t-1$ 时刻网络中的记忆单元，传入 t 时刻的网络之后，第一步操作是决定它的遗忘程度，将 t 时刻前面的记忆状态乘上一个 $0 \sim 1$ 的系数进行衰减，接着加上 t 时刻学到的记忆作为更新之后的记忆传出网络，作为 $t+1$ 时刻网络的记忆单元。其中 $t-1$ 时刻网络记忆的衰减系数是通过 t 时刻网络的输入和 $t-1$ 网络的输出来确定的，t 时刻网络学到的记忆也是根据 t 时刻网络的输入和 $t-1$ 时刻网络的输出得到的，下面会具体介绍如何计算出结果。

标准的循环神经网络内部只有一个简单的层结构，而 LSTM 内部有 4 个层结构，下面来依次解释它们内部的运算方式以及如何表示上面所说的输入门、遗忘门和输出门。

首先是整个网络结构中最重要的遗忘门，如图 5.12所示。

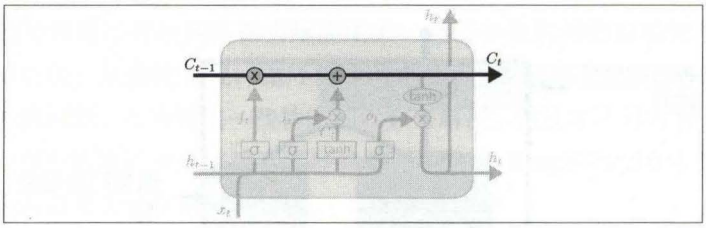


图 5.12　网络结构的遗忘门

首先介绍记忆的衰减系数是如何得到的，计算过程如图5.13所示。

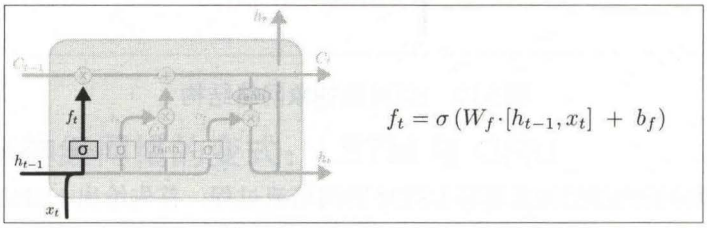


图 5.13　衰减系数计算过程

首先将 $t-1$ 时刻的网络输出 h_{t-1} 和这一步的网络输入 x_t 结合起来，然后作用线性变换 $W_f \cdot [h_{t-1}, x_t] + b_f$，最后再经过 sigmoid 激活函数，将结果映射到 $0 \sim 1$ 作为记忆的衰减系数，记作 f_t。可以看到网络具体要保留多少记忆是由前一时刻的输出和这一时刻的输入共同决定的。

接着介绍 t 时刻学到的记忆是如何计算的，过程如图 5.14所示。

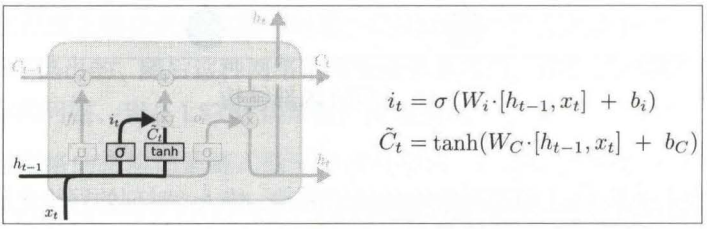


图 5.14　t 时刻学到记忆的计算过程

首先对该时刻学到的记忆，对它应用一个衰减系数，这个系数应用跟上面的方式相同，再使用线性变换，然后使用 sigmoid 激活函数，将结果映射到 $0 \sim 1$ 之间，这个结果作为当前学习到记忆的衰减系数，记作 i_t。当前状态学习到的记忆 $\tilde{C}_t$ 是通过线性变换 $W_C \cdot [h_{t-1}, x_t] + b_C$ 和 tanh 激活函数得到的。

最后将 $t-1$ 时刻的衰减系数 f_t 乘 $t-1$ 时刻的记忆 C_{t-1}，加上该时刻 t 下学到的记忆 $\tilde{C}_t$ 乘以它对应的衰减系数 i_t，这样便得到了 t 时刻下的记忆状态 C_t，可以用图5.15来显示具体的计算过程。

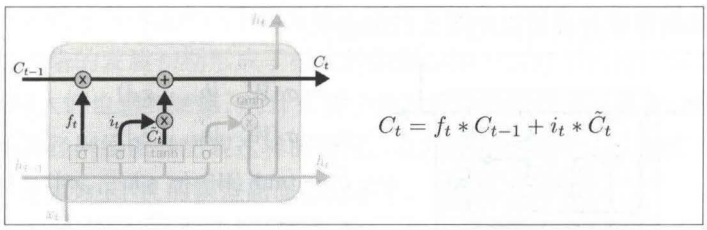


图 5.15 C_t 计算过程

上面的过程就是 LSTM 中的输入门和遗忘门，最后介绍输出门。

当前时刻 t 的网络输出 h_t 取决于当前时刻 t 的记忆状态 C_t 和 t 时刻的输入 x_t、$t-1$ 时刻的输出 h_{t-1}，具体的计算过程如图 5.16所示。

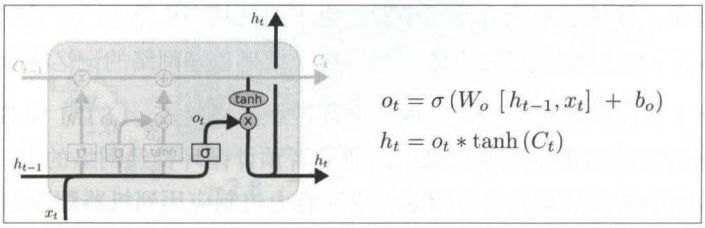


图 5.16 o_t 与 h_t 的计算过程

首先使用类似于计算记忆衰减系数的方法计算得到输出门的系数 o_t，这个系数决定了输出的多少，最后网络的输出由 $h_t = o_t \times tanh(C_t)$ 得到，这就是输出门如何控制网络输出的原理。

上面介绍了 LSTM 内部的网络结构和计算方式，它和传统循环神经网络最大的不同在于它使用了输入门、遗忘门和输出门来控制网络实现长时记忆的功能。实际上在每篇使用 LSTM 的论文中，作者都使用了和标准 LSTM 稍微不同的版本，这些变式的网络结构并不存在哪一个是最好的说法，针对特定的任务，特定的网络结构往往表现得更好。

接下来介绍和 LSTM 比，有着更大区别的网络结构——GRU。

5.2.2 GRU

GRU 是 Gated Recurrent Unit 的缩写，由 Cho 在 2014 年提出。GRU 和 LSTM 最大的不同在于 GRU 将遗忘门和输入门合成了一个“更新门”，同时网络不再额外给出记忆状态 C_t，而是将输出结果 h_t 作为记忆状态不断向后循环传递，网络的输入和输出都变得特别简单。

GRU 具体的计算过程可以用图 5.17来表示。

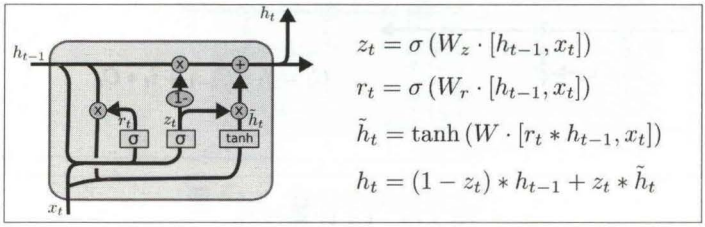


图 5.17　GRU 计算过程

上面给出了 GRU 具体的计算过程，就不再赘述了。它和 LSTM 本质上是相同的，将上一时刻 $t-1$ 的输出 h_{t-1} 和当前 t 时刻的输入 x_t 结合起来计算各种衰减系数，略微不同的地方是，线性变换没有使用偏置，由于记忆状态也是 h_{t-1}，所以直接对它进行更新就可以了，最后输出网络的结果 h_t，这个结果也是网络的记忆状态。

以上了解了目前使用最为广泛的两种循环神经网络变式：LSTM 和 GRU，它们为循环神经网络的发展做出了重要贡献，而研究者预计循环神经网络的下一步发展将是注意力机制，本书的实战部分将介绍如何将注意力机制应用到机器翻译中。

5.2.3　收敛性问题

虽然循环神经网络由最初的标准形式发展出了很多变式，比如 LSTM 和 GRU，但是这些网络都无可避免地存在一个问题，那就是收敛性。

如果写了一个简单的 LSTM 网络去训练数据，你会发现 loss 并不会按照想象的方式下降，如果运气好的话能够得到一直下降的 loss，但是大多数情况下 Loss 都是在乱跳着的，如图 5.18所示。

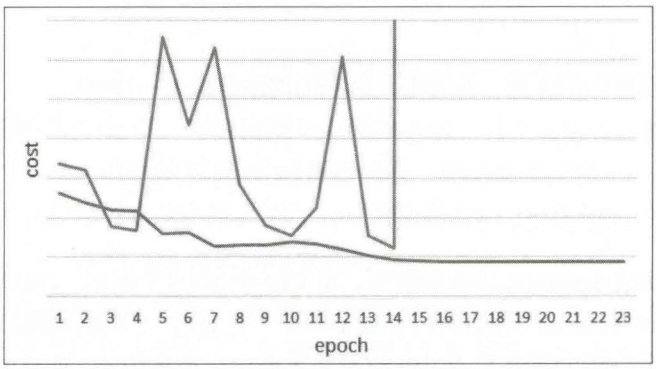


图 5.18　loss 乱跳

运气好的话能够得到蓝色的下降曲线，运气不好的话就会就得到绿色的下降曲线，这在循环神经网络的发展初期形成了很大的阻碍，因为这种不稳定的训练过程是没办法使用的。同时人们也感到疑惑，为什么基于卷积的神经网络不会出现这种跳跃的 loss，而基于循环的神经网络就会出现这样的情况。在随后的研究中，人们发现出现这种情况的根本原因是因为 RNN 的误差曲面粗糙不平，如图5.19所示。

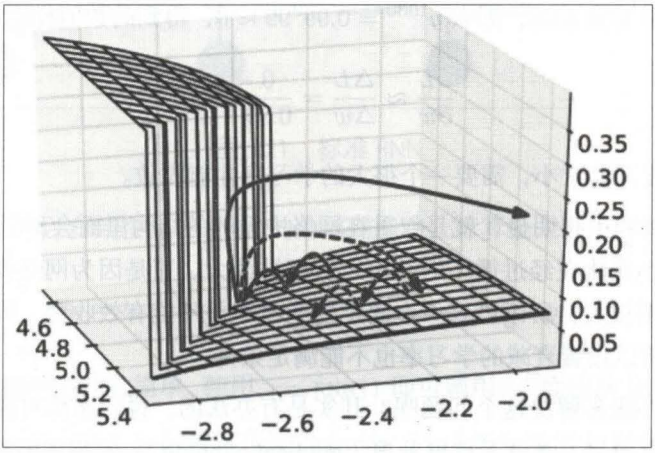


图 5.19 RNN 误差曲面粗糙不平

误差曲面上存在很多陡峭的斜坡，斜坡上误差的变化率特别大，正是这个原因导致道路 loss 曲线在不断地跳跃。对于这种基于循环的网络结构的不规则、陡峭的斜坡曲面，能够采用什么方法对网络进行训练呢？可能有的人会说只需使用比较小的学习率就可以了，下面用一个例子来说明这种方法是不可行的。

首先引入一个非常简单的循环网络，如图5.20所示，隐藏状态的权重是 w，其余权重都是 1，网络输入的序列长度是 1000，除了第一个是 1 之外，其余的都是 0，网络上的 $y^1 \cdots y^{1000}$ 表示输出，计算得到序列最后的输出是 w^{999}。

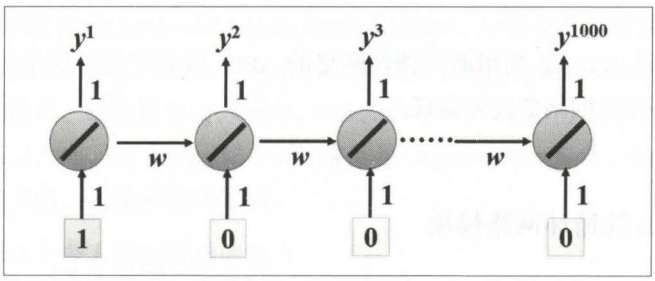


图 5.20 简单的循环网络

对于这样一个结构，如果 $w = 1$，那么输出 $y^{1000} = 1^{999} = 1$，如果 $w = 1.01$，那

么输出 $y^{1000} = 1.01^{999} \approx 20000$，所以计算出它的数值梯度如下：

$$\frac{\partial L}{\partial w} \approx \frac{\Delta L}{\Delta w} = \frac{20000 - 1}{1.01 - 1} \tag{5.1}$$

可以看到，这里的梯度非常大，所以需要非常小的学习率才能够收敛。

但是，如果 $w = 0.99$，那么 $y^{1000} = 0.99^{9}99 \approx 0$，得到的数值梯度如下：

$$\frac{\partial L}{\partial w} \approx \frac{\Delta L}{\Delta w} = \frac{0 - 1}{0.99 - 1} \tag{5.2}$$

这样得到的梯度又非常小，需要一个很大的学习率才能收敛。

这个问题的原因很明显，就是权重在网络中循环的结构里面会不断地被重复使用，那么梯度的微小变化在经过循环的结构之后都被放大。正是因为网络在训练的过程中梯度的变化范围很大，所以设置一个固定的学习率并不能有效收敛，同时梯度的变化并没有规律，所以设置衰减的学习率也不能满足条件。

是否没有办法来解决这个问题呢？其实是有办法的，说出来你可能会觉得太简单了，但是确实有效果，那就是梯度裁剪（gradient clipping）。使用梯度裁剪能够将大的梯度裁掉，这样就能够在一定程度上避免收敛不好的问题。

当然，现在能用很多别的办法解决这个问题，这里只是提出循环神经网络确实存在着收敛性问题，有兴趣的读者可以查阅一些近年的论文。

5.3 循环神经网络的 PyTorch 实现

前一节介绍了循环神经网络的理论和具体计算，这些复杂的计算并不需要从头去写，PyTorch 中早已集成好了一切供我们调用，下面这个部分将介绍循环神经网络的 PyTorch 实现。

首先先介绍 PyTorch 调用循环神经网络的 API，接着用两个简单的例子具体说明 PyTorch 中循环神经网络的具体实现。

5.3.1 PyTorch 的循环网络模块

下面会分别介绍标准 RNN、LSTM 和 GRU 模块在 PyTorch 中的调用。

1. 标准 RNN

先给出标准 RNN 的示意图，如图 5.21所示，按照图 5.21 来讲解 PyTorch 中的 API。

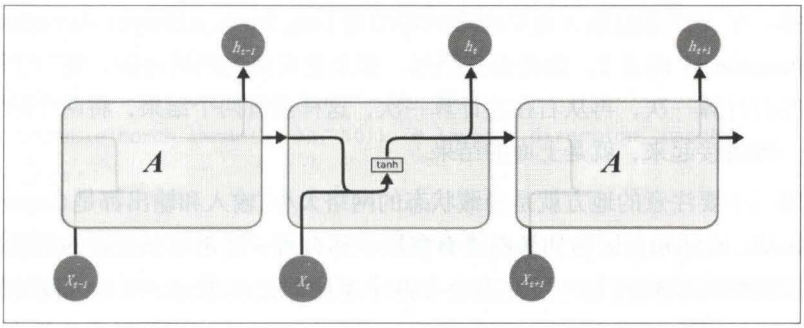


图 5.21 标准 RNN

从图 5.21 中可以看到在标准 RNN 的内部网络中，计算公式如下：

$$h_t = tanh(w_{ih} * x_t + b_{ih} + w_{hh} * h_{t-1} + b_{hh}) \tag{5.3}$$

在 PyTorch 中的调用也非常简单，使用 `nn.RNN()` 即可调用，下面依次介绍其中的参数。

`input_size` 表示输入 x_t 的特征维度；`hidden_size` 表示输出 h_t 的特征维度；`num_layers` 表示网络层数，默认是 1 层；`nonlinearity` 表示非线性激活函数的选择，默认是 tanh，可以选择 relu；`bias` 表示是否使用偏置，默认是 True；`batch_first` 这个参数是决定网络输入的维度顺序，默认网络输入是按照 (seq, batch, feature) 输入的，也就是序列长度放在最前面，然后是批量，最后是特征维度，如果这个参数设置为 True，那么顺序就变为 (batch, seq, feature)；`dropout` 这个参数接受一个 0 ～ 1 的数值，会在网络中除了最后一层之外的其他输出层加上 dropout 层；`bidirectional` 默认是 False，如果设置为 True，就是双向循环神经网络的结构。

接着再介绍网络接收的输入和输出。网络会接收一个序列输入 x_t 和记忆输入 h_0，x_t 的维度是 (seq, batch, feature)，分别表示序列长度、批量和输入的特征维度，h_0 也叫隐藏状态，它的维度是 ($layers * direction$, batch, hidden)，分别表示层数乘方向（如果是单向，就是 1，如果是双向就是 2）、批量和输出的维度。网络会输出 output 和 h_n，output 表示网络实际的输出，维度是 (seq，batch，$hidden * direction$)，分别表示序列长度、批量和输出维度乘上方向，h_n 表示记忆单元，维度是 ($layer * direction$，$batch$，$hidden$)，分别表示层数乘方向、批量和输出维度。

下面来解释几个令人感到疑惑的地方。

- 第一个要注意的地方就是网络的输入和前面讲过的卷积网络有些不同，因为卷积神经网络的输入将 batch 放在前面，而在循环神经网络中将 batch 放在中间，当然可以使用 `batch_first=True` 让 batch 放在前面。

- 第二个要注意的地方就是网络的输出是 (seq, batch, $hidden * direction$)，这里 direction=1 或者 2，前面也介绍过，如果是双向的网络结构，相当于网络从左往右计算一次，再从右往左计算一次，这样会有两个结果，将两个结果按最后一维拼接起来，就是上面的结果。
- 第三个要注意的地方就是隐藏状态的网络大小、输入和输出都是 ($layer*direction$, batch, hidden)，因为如果网络有多层，那么每一层都有一个新的记忆单元，而双向网络结构在每一层左右会有两个不同的记忆单元，所以维度的第一位是 $layer * direction$。

下面用 PyTorch 实现简单的循环神经网络，并验证上面所介绍的内容。

首先建立一个简单的循环神经网络：输入维度是 20、输出维度是 50、两层的单向网络。

```
basic_rnn = nn.RNN(input_size=20, hidden_size=50, num_layers=2)
```

对于式（5.1）中的每个权重，都能够通过如图 5.22所示的方式直接访问。

对于定义好的 RNN，可以通过 `weight_ih_l0` 来访问第一层中的 w_{ih}，因为输入 x_t 是 20 维，输出是 50 维，所以 w_{ih} 是一个 50×20 的向量，另外要访问第二层网络可以使用 `weight_ih_l1`。对于 w_{hh}，可以用 `weight_hh_l0` 来访问，而 b_{ih} 则可以用 `bias_ih_l0` 来访问。当然可以对它进行自定义的初始化，只需要记得这些参数都是 Variable，取出它们的 data，对它进行自定的初始化即可。

```
basic_rnn.weight_ih_l0

Parameter containing:
-4.4217e-02 -1.1636e-02 -1.0950e-01  ...   3.0889e-02 -5.3554e-02  1.3470e-02
 2.0199e-02  2.6346e-02 -1.2236e-01  ...   3.0369e-02  3.1875e-02  7.5414e-02
-7.3592e-02 -1.0765e-01 -6.4064e-02  ...  -1.4163e-02 -2.7480e-02 -7.8861e-02
                ...                   ⋱                ...
 1.3970e-01  5.2258e-02 -1.5994e-02  ...   7.2413e-02 -1.1146e-01 -4.2005e-02
 1.9304e-02  1.2416e-01  4.3513e-02  ...  -6.9903e-02 -1.9987e-02  9.1069e-02
-1.2357e-01  1.3164e-01  4.9964e-02  ...  -1.2016e-01  1.1797e-01 -7.6756e-02
[torch.FloatTensor of size 50x20]

basic_rnn.weight_hh_l0

Parameter containing:
 0.0022  0.0818  0.0834  ...  -0.1024 -0.0085 -0.0421
-0.0973  0.1049 -0.0598  ...  -0.0427  0.0391 -0.0953
-0.0930 -0.0776  0.1303  ...   0.0230 -0.1142  0.1007
    ...                   ⋱          ...
-0.0447 -0.1351 -0.0234  ...   0.0948  0.0350  0.0472
 0.0477  0.1165 -0.0886  ...  -0.0022  0.1095 -0.0051
-0.0308  0.0166  0.0746  ...  -0.0695  0.0498  0.0692
[torch.FloatTensor of size 50x50]
```

图 5.22　直接访问方式

接着将输入传入网络，验证得到的输出是否如前面介绍的一样。首先随机初始化输入和隐藏状态如下，输入是一个长为 100、批量为 32、维度为 20 的张量，隐藏状态

的维度按照网络的需求定义如下：

```
toy_input = Variable(torch.randn(100, 32, 20))
h_0 = Variable(torch.randn(2, 32, 50))  # layer * direction, batch,
hidden_size
```

然后将输入和隐藏状态传入网络，得到输出和更新之后的隐藏状态，输出的长度是 100、批量是 32、维度是 50，和前面介绍的一致，而更新之后的隐藏状态和输入的隐藏状态也是大小相同的。

```
toy_output, h_n = basic_rnn(toy_input, h_0)
print(toy_output.size())  # seq, batch, hidden_size
print(h_n.size())  # layer * direction, batch, hidden_size
```

得到它们的维度分别是 (100, 32, 50) 和 (2, 32, 50)。

如果在传入网络的时候不特别注明隐藏状态 h_0，那么初始的隐藏状态默认参数全是 0，当然也可以用上面的方式来自定义隐藏状态的初始化。

2.LSTM

LSTM 在本质上和标准 RNN 是一样的，只不过 LSTM 内部的计算更加复杂、参数更多、输入和输出的数目也更多。

图 5.23所示的就是 LSTM 的计算过程，在 PyTorch 中调用也非常简单，使用 `nn.LSTM()` 即可，里面的参数和标准 RNN 中的参数一模一样，就不再赘述，下面介绍 LSTM 与标准 RNN 不同的地方。

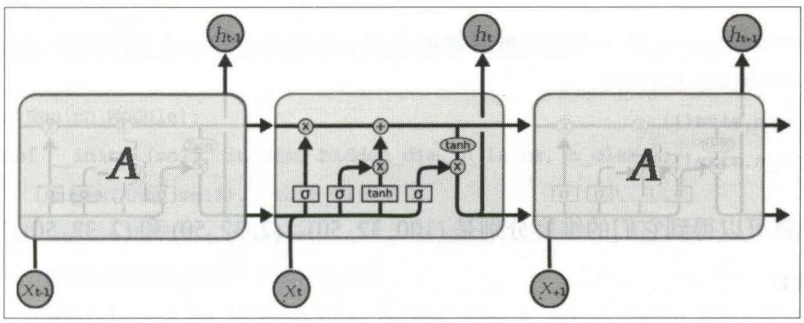


图 5.23 LSTM 的计算过程

首先，LSTM 的参数比标准 RNN 多，但是访问的方式仍然是相同的，使用 `weight_ih_l0` 即可，只是里面的维度和标准 RNN 不再相同，它是标准 RNN 维度的 4 倍，因为 LSTM 中间比标准 RNN 多了三个线性变换，多的三个线性变换的权重拼在一起，所

以一共是 4 倍，同理偏置也将是 4 倍。换句话说，LSTM 里面做了 4 个类似标准 RNN 所做的运算，所以参数个数是标准 RNN 的 4 倍。

其次，LSTM 的输入也不再只有序列输入和隐藏状态，从图 5.23 也可以看出，隐藏状态除了 h_0 以外，还多了一个 C_0，它们合在一起成为网络的隐藏状态，而且它们的大小完全一样，就是 ($layer * direction$, batch, hidden)，当然输出也会有 h_0 和 C_0。

下面定义一个和标准 RNN 输入输出相同的 LSTM 如下：

```
lstm = nn.LSTM(input_size=20, hidden_size=50, num_layers=2)
```

然后可以访问其中的参数，使用 `weight_ih_l0` 得到如图5.24所示的结果。

```
lstm.weight_ih_l0

Parameter containing:
 0.0648 -0.0786 -0.1359  ...   0.1099  0.0977 -0.1010
-0.0857 -0.1410 -0.0689  ...  -0.0929 -0.0275 -0.0984
-0.0641  0.1151 -0.0781  ...  -0.0754  0.1070  0.1384
           ...                    ⋱              ...
 0.0431 -0.0398  0.1186  ...   0.0470  0.1370 -0.0892
-0.0578 -0.1063  0.1066  ...  -0.0205 -0.0217  0.1318
-0.0685  0.1297 -0.0677  ...   0.0851 -0.1152 -0.0856
[torch.FloatTensor of size 200x20]
```

图 5.24　LSTM 的计算结果

可以看到，参数的大小变成了 (50×4, 20)，确实变成了标准 RNN 的 4 倍。

传入输入，这次没有传入隐藏状态，那么默认就会传入参数全是 0 的隐藏状态 h_0 和 C_0，同样可以得到结果如下：

```
lstm_out, (h_n, c_n) = lstm(toy_input)
print(lstm_out.size())
print(h_n.size())
print(c_n.size())
```

这样就可以得到它们的维度分别是 (100, 32, 50)、(2, 32, 50) 和 (2, 32, 50)。

3.GRU

GRU 本质上和 LSTM 是一样的，它的网络计算图前面也具体介绍了，这里简单介绍它与 LSTM 不同的地方。

首先它的隐藏状态参数不再是标准 RNN 的 4 倍，而是 3 倍，这是由于它内部计算结构确定的。同时网络的隐藏状态也不再是 h_0 和 C_0，而是只有 h_0，这里从网络的计算图中也能看出来，其余部分和 LSTM 完全一样，就不再赘述。

除此之外，PyTorch 中还提供了 `RNNCell`、`LSTMCell`、`GRUCell`，这三个与前面介绍的有什么区别呢？

这三个分别是上面介绍的三个函数的单步版本，也就是说它们的输入不再是一个序列，而是一个序列中的一步，也可以说是循环神经网络的一个循环，在序列的应用上更加灵活，因为序列中每一步都是手动实现的，能够在基础上添加更多自定义的操作，这个部分就不再介绍，感兴趣的读者可以自行查阅文档。

5.3.2 实例介绍

前面的部分介绍了 RNN、LSTM 和 GRU 的基本概念，以及在 PyTorch 中如何调用，下面用两个例子来熟悉整体的操作。

图片分类

循环神经网络特别适用于序列数据，那么对于图片类型的数据，是不是循环神经网络就无办法处理了呢？其实不是这样的，仍然可以用循环神经网络进行图片分类，下面就对 MNIST 手写数字进行分类。

首先需要将图片数据转化为一个序列数据，MNIST 手写数字的图片大小是 28×28，那么可以将每张图片看作是长为 28 的序列，序列中的每个元素的特征维度是 28，这样就将图片变成了一个序列。同时考虑到循环神经网络的记忆性，所以图片从左往右输入网络的时候，网络可以记忆住前面观察到的东西，也就是说一张图片虽然被切割成了 28 份，但是网络能够通过记住前面的部分，同时和后面的部分结合得到最后预测数字的输出结果，所以从理论上而言是行得通的。

下面介绍如何定义一个用于图片分类的循环神经网络。

```
class Rnn(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_layer, n_class):
        super(Rnn, self).__init__()
        self.n_layer = n_layer
        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(in_dim, hidden_dim, n_layer, batch_first=True)
        self.classifier = nn.Linear(hidden_dim, n_class)

    def forward(self, x):
        # h0 = Variable(torch.zeros(self.n_layer, x.size(1),
        #   self.hidden_dim)).cuda()
```

```
        # c0 = Variable(torch.zeros(self.n_layer, x.size(1),
        #    self.hidden_dim)).cuda()
        out, _ = self.lstm(x)
        out = out[:, -1, :]
        out = self.classifier(out)
        return out
```

上面的定义中网络主要由 LSTM 网络和线性网络构成，LSTM 网络接受图片序列，线性网络将它输出成最后的概率向量。因为处理的是图像数据，往往是 batch 放在前面，所以在 LSTM 的定义中，使用了 `batch_first=True`，这样网络的输出也是 batch 在前面。另外在 `forward` 要注意一个细节，out=out[:, -1, :]，这是因为循环神经网络的输出也是一个序列，这一行代码是取出输出序列中的最后一个，在应用线性层作为最后的输出结果。

经过上面的介绍，整个过程就变得清晰了，而网络的整个训练过程跟之前一模一样，就不再赘述。最后训练 20 次，得到如图 5.25所示的结果。

```
epoch 19
**********
[19/20] Loss: 0.008209, Acc: 0.997400
[19/20] Loss: 0.010693, Acc: 0.996633
Finish 19 epoch, Loss: 0.010693, Acc: 0.996633
Test Loss: 0.044867, Acc: 0.987600

epoch 20
**********
[20/20] Loss: 0.007949, Acc: 0.997567
[20/20] Loss: 0.010148, Acc: 0.996817
Finish 20 epoch, Loss: 0.010148, Acc: 0.996817
Test Loss: 0.039617, Acc: 0.990800
```

图 5.25　训练结果

可以看到在 MNIST 数据集上能够达到 99% 的验证集准确率，应该是比较高了。虽然在一个简单的图片数据集上能够达到相对满意的效果，但是循环神经网络还是不适合处理图片类型的数据：

- 第一个原因是图片并没有很强的序列关系，图片中的信息可以从左往右看，也可以从右往左看，甚至可以跳着随机看，不管是什么样的方式都能够完整地理解图片信息；
- 第二个原因是循环神经网络传递的时候，必须前面一个数据计算结束才能进行后面一个数据的计算，这对于大图片而言是很慢的，但是卷积神经网络并不需要这样，因为它能够并行，在每一层卷积中，并不需要等待第一个卷积做完才能做第二个卷积，整体是可以同时进行的。

下面就来介绍循环神经网络真正适用的场景——序列预测。

对序列数据而言，因为它有着时序性，即前面的数据对后面数据有影响，所以LSTM的记忆性能够适用于这种场景。

读入的数据是2010年的飞机月流量，可视化后得到如图5.26所示的结果。

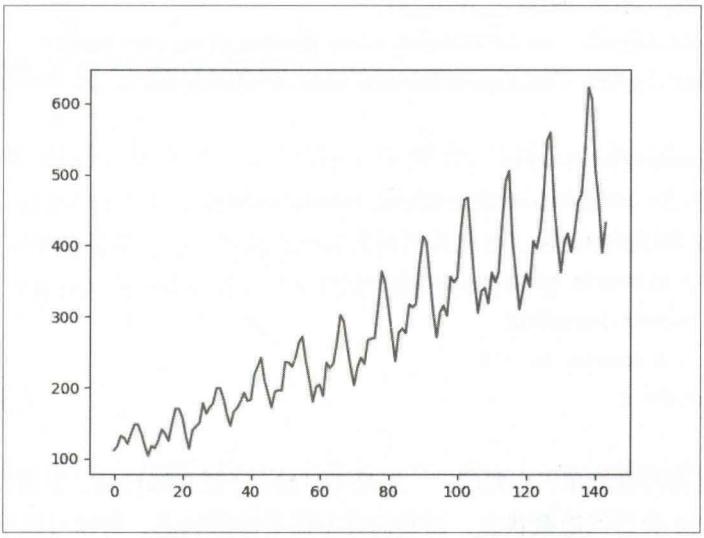


图 5.26 2010年的飞机月流量

首先需要将数据标准化到0～1之间，这是预处理的标准步骤，接着开始建立数据集。因为给出的是纯数据，没有任何特征，所以希望使用前几个月的流量预测当前月的流量，所以可以建立以下数据集：

```
def create_dataset(dataset, look_back=2):
    dataX, dataY = [], []
    for i in range(len(dataset) - look_back):
        a = dataset[i:(i + look_back)]
        dataX.append(a)
        dataY.append(dataset[i + look_back])
    return np.array(dataX), np.array(dataY)
```

这里使用前两个月的流量数据预测当前月的流量，也可以设置为前三个月。然后需要将数据分为训练集和测试集，这里简单地将前面几年的数据作为训练集，后面两年的数据作为测试集。接着需要再处理数据，因为网络读入的数据维度是(seq, batch, input)，所以重新排列数据就可以了，这里的batch是1，因为只有一个序列，input就是预测依据的月份数，这里设置为2，序列长度就是前面划分好的训练集的序列长度。

接着建立简单的循环神经网络，这里使用非常流行的LSTM：

```
class lstm(nn.Module):
    def __init__(self, input_size=2, hidden_size=4, output_size=1,
                 num_layer=2):
        super(lstm, self).__init__()
        self.layer1 = nn.LSTM(input_size, hidden_size, num_layer)
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x, _ = self.layer1(x)  # seq, batch, hidden
        s, b, h = x.size()
        x = x.view(s * b, h)
        x = self.layer2(x)
        x = x.view(s, b, -1)
        return x
```

网络的定义特别简单，有两层：一层是 LSTM，一层是线性层，主要介绍向前传播的部分。LSTM 会返回隐藏状态，而我们并不需要隐藏状态，所以可以不保存这个状态，接着需要使用 `view` 来重新排列，因为 `nn.Linear` 不接受三维的输入，只接受两维的输入，所以我们可以将前面两维先合并到一起，然后经过线性层之后再把它们分开，最后输出结果。另外，网络输入的维度是根据前面建立数据来确定的，建立数据时依赖前面两个月的流量来预测第三个月的流量，那么输入的维数就是 2，LSTM 网络的输出可以任意定义，和后面的线性层适配即可。

最后使用 Adam 训练 100 次，得到了如图 5.27所示的结果。

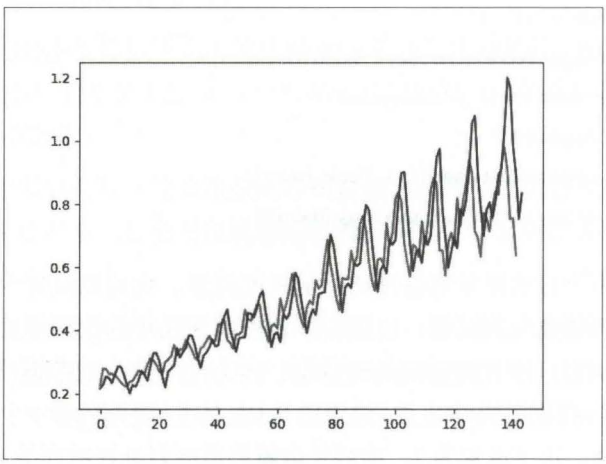


图 5.27　Adam 训练 100 次的结果

其中蓝色是真实的数据，红色是预测的数据，可以看到使用 LSTM 能够得到相对近似的预测结果，如果只使用线性回归，并不能达到比较理想的效果，这个例子也说明了循环神经网络相比序列数据性能更好。

5.4 自然语言处理的应用

前面的部分具体介绍了如何在 PyTorch 中调用循环神经网络，同时列举了两个例子：一个是 MNIST 手写数字分类，另一个是飞机流量的时间序列，通过这两个例子说明循环神经网络的简单应用，但是循环神经网络目前在自然语言处理中应用最为火热，所以这一小节将介绍自然语言处理中如何使用循环神经网络。

5.4.1 词嵌入

首先介绍自然语言处理中的第一个概念——词嵌入（word embedding），也可以称为词向量。

图像分类问题会使用 one-hot 编码，比如一共有五类，那么属于第二类的话，它的编码就是 (0, 1, 0, 0, 0)，对于分类问题，这样当然特别简明。但是在自然语言处理中，因为单词的数目过多，这样做就行不通了，比如有 10000 个不同的词，那么使用 one-hot 这样的方式来定义，效率就特别低，每个单词都是 10000 维的向量，其中只有一位是 1，其余都是 0，特别占用内存。除此之外，也不能体现单词的词性，因为每一个单词都是 one-hot，虽然有些单词在语义上会更加接近，但是 one-hot 没办法体现这个特点，所以必须使用另外一种方式定义每一个单词，这就引出了词嵌入。

词嵌入到底是什么意思呢？其实很简单，对于每个词，可以使用一个高维向量去表示它，这里的高维向量和 one-hot 的区别在于，这个向量不再是 0 和 1 的形式，向量的每一位都是一些实数，而这些实数隐含着这个单词的某种属性。这样解释可能不太直观，先举四个例子，下面有 4 段话：

（1）The cat likes playing ball.

（2）The kitty likes playing wool.

（3）The dog likes playing ball.

（4）The boy likes playing ball.

重点分析里面的 4 个词，cat、kitty、dog 和 boy。如果使用 one-hot，那么 cat 就可以表示成 (1, 0, 0, 0)，kitty 就可以表示成 (0, 1, 0, 0)，但是 cat 和 kitty 其实都表示小猫，所以这两个词语义是接近的，但是 one-hot 并不能体现这个特点。

下面使用词嵌入的方式来表示这 4 个词，假如使用一个二维向量 (a, b) 来表示一个词，其中 a，b 分别代表这个词的一种属性，比如 a 代表是否喜欢玩球，b 代表是否喜欢玩毛线，并且这个数值越大表示越喜欢，这样就能够定义每一个词的词嵌入，并且通过这个来区分语义，下面来解释一下原因。

对于 cat，可以定义它的词嵌入是 (-1, 4)，因为它不喜欢玩球，喜欢玩毛线；而对于 kitty，它的词嵌入可以定义为 (-2, 5)；那么对于 dog，它的词嵌入就是 (3, -2)，因为它喜欢玩球，不喜欢玩毛线；最后对于 boy，它的词向量就是 (-2, -3)，因为这两样东西他都不喜欢。定义好了这样的词嵌入，怎么去定义它们之间的语义相似度呢？可以通过词向量之间的夹角来定义它们的相似度。下面先将每个词向量都在坐标系中表示出来，如图 5.28所示。

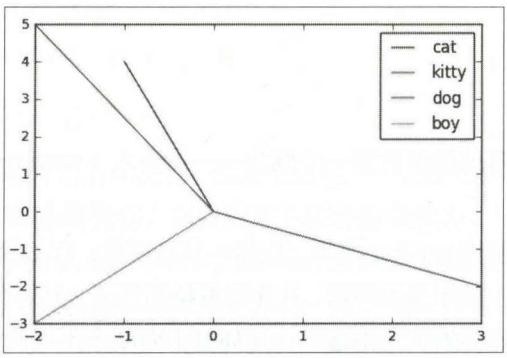


图 5.28　不同词向量的夹角

图 5.28 就显示出了不同词向量之间的夹角，可以发现 kitty 和 cat 的夹角更小，所以它们更加相似的，而 dog 和 boy 之间夹角很大，所以它们不相似。

通过这样一个简单的例子能够看出词嵌入对于单词的表示具有很好的优势，但是问题来了，对于一个词，怎么知道如何去定义它的词嵌入？如果向量的维数只有 5 维，可能还能定义出来，如果向量的维数是 100 维，那么怎么知道每一维体是多少呢？

这个问题可以交给神经网络去解决，只需要定义我们想要的维度，比如 100 维，神经网络就会自己去更新每个词嵌入中的元素。而之前介绍过词嵌入的每个元素表示一种属性，当然对于维数比较低的时候，可能我们能够推断出每一维具体的属性含义，然而维度比较高之后，我们并不需要关心每一维到底代表着什么含义，因为每一维都是网络自己学习出来的属性，只需要知道词向量的夹角越小，表示它们之间的语义更加接近就可以了。这就好比卷积网络会对一张图片提取出很厚的特征图，并不需要关心网络提取出来的特征到底是什么，只需要知道抽象的特征能够帮助我们分类图像就可以了。

5.4.2 词嵌入的 PyTorch 实现

词嵌入在 PyTorch 中是如何实现的呢？下面来具体实现一下。

PyTorch 中的词嵌入是通过函数 `nn.Embedding(m, n)` 来实现的，其中 m 表示所有的单词数目，n 表示词嵌入的维度，下面举一个例子：

```
word_to_ix = {'hello': 0, 'world': 1}
embeds = nn.Embedding(2, 5)
hello_idx = torch.LongTensor([word_to_ix['hello']])
hello_idx = Variable(hello_idx)
hello_embed = embeds(hello_idx)
print(hello_embed)
```

上面就是输出的 hello 的词嵌入，下面来解释一下代码。首先需要给每个单词建立一个对应下标，这样每个单词都可以用一个数字去表示，比如需要 hello 的时候，就可以用 0 来表示，用这种方式，访问每个词会特别方便。接着是词嵌入的定义 `nn.Embedding(2, 5)`，如上面介绍过的，表示有两个词，每个词向量是 5 维，也就是一个 2×5 的矩阵，只不过矩阵中的元素是可以被学习更新的，所以如果有 1000 个词，每个词向量希望是 100 维，就可以这样定义词嵌入 `nn.Embedding(1000, 100)`。访问每一个词的词向量需要将 tensor 转换成 Variable，因为词向量也是网络中更新的参数，所以在计算图中，需要通过 Variable 去访问。另外这里的词向量只是初始的词向量，并没有经过学习更新，需要建立神经网络优化更新，修改词向量里面的参数使得词向量能够表示不同的词，且语义相近的词能够有更小的夹角。

以上介绍了词嵌入在 PyTorch 中是如何实现的，下一节将介绍词嵌入是如何更新的，以及它如何结合 N Gram 语言模型进行预测。

5.4.3 N Gram 模型

首先介绍 N Gram 模型的原理和它要解决的问题。在一篇文章中，每一句话都是由很多单词组成的，而且这些单词的排列顺序也是非常重要的。在一句话中，是否可以由前面几个词来预测这些词后面的一个单词？比如在“I lived in France for 10 years, I can speak _.”这句话中，我们希望能够预测最后这个词是 French。

知道想要解决的问题后，就可以引出 N Gram 语言模型了。对于一句话 T，它由

$w_1, w_2, \cdots w_n$ 这 n 个词构成，可以得到下面的公式：

$$P(T) = P(w_1)P(w_2|w_1)P(w_3|w_2w_1)\cdots P(w_n|w_{n-1}w_{n-2}\cdots w_2w_1) \tag{5.4}$$

但是这样的一个模型存在着一些缺陷，比如参数空间过大，预测一个词需要前面所有的词作为条件来计算条件概率，所以在实际中没办法使用。为了解决这个问题，引入了马尔科夫假设，也就是说这个单词只与前面的几个词有关系，并不是和前面所有的词都有关系，有了这个假设，就能够在实际中使用 N Gram 模型了。

对于这个条件概率，传统的方法是统计语料中每个单词出现的频率，据此来估计这个条件概率，这里使用词嵌入的办法，直接在语料中计算这个条件概率，然后最大化条件概率从而优化词向量，据此进行预测。

5.4.4 单词预测的 PyTorch 实现

首先给出一段文章作为训练集：

```
CONTEXT_SIZE = 2
EMBEDDING_DIM = 10
# We will use Shakespeare Sonnet 2
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""".split()
```

CONTEXT_SIZE 表示想由前面的几个单词来预测这个单词，这里设置为 2，就是说我们希望通过这个单词的前两个单词来预测这一个单词，EMBEDDING_DIM 表示词嵌入的维数。

接着建立训练集，遍历所有语料来创建，将数据整理好，需要将单词分三个组，每个组前两个作为传入的数据，而最后一个作为预测的结果。

```
trigram = [((test_sentence[i], test_sentence[i+1]), test_sentence[i+2])
           for i in range(len(test_sentence)-2)]
```

将每个单词编码，即用数字来表示每个单词，只有这样才能够传入 nn.Embedding 得到词向量。

```
vocb = set(test_sentence) # 通过set将重复的单词去掉
word_to_idx = {word: i for i, word in enumerate(vocb)}
idx_to_word = {word_to_idx[word]: word for word in word_to_idx}
```

然后可以定义 N Gram 模型如下：

```
class NgramModel(nn.Module):
    def __init__(self, vocb_size, context_size, n_dim):
        super(NgramModel, self).__init__()
        self.n_word = vocb_size
        self.embedding = nn.Embedding(self.n_word, n_dim)
        self.linear1 = nn.Linear(context_size*n_dim, 128)
        self.linear2 = nn.Linear(128, self.n_word)

    def forward(self, x):
        emb = self.embedding(x)
        emb = emb.view(1, -1)
        out = self.linear1(emb)
        out = F.relu(out)
        out = self.linear2(out)
        log_prob = F.log_softmax(out)
        return log_prob
```

模型需要传入的参数有三个，分别是所有的单词数、预测单词所依赖的单词数、即 CONTEXT_SIZE 和词向量的维度。网络在向前传播中，首先传入单词得到词向量，模型是根据前面两个词预测第三个词的，所以需要传入两个词，得到的词向量是 (2, 100)，然后将词向量展开成 (1, 200)，接着经过线性变换，经过 relu 激活函数，再经过一个线性变换，输出的维数是单词总数，最后经过一个 log softmax 激活函数得到概率分布，最

大化条件概率，可以用下面的公式表示：

$$-log(p(w_i|C)) = -log(Softmax(A(\sum_{w \in Cq_w}) + b)) \tag{5.5}$$

在网络的训练中，不仅会更新线性层的参数，还会更新词嵌入中的参数，训练 100 次模型，可以发现 loss 已经降到了 0.37，也可以通过预测来检测模型是否有效：

```
word, label = trigram[3]
word = Variable(torch.LongTensor([word_to_idx[i] for i in word]))
out = ngrammodel(word)
_, predict_label = torch.max(out, 1)
predict_word = idx_to_word[predict_label.data[0][0]]
print('real word is {}, predict word is {}'.format(label, predict_word))
```

运行上面的代码，可以发现真实的单词跟预测的单词都是一样的，虽然这是在训练集上，但是在一定程度上也说明这个小模型能够处理 N Gram 模型的问题。

上面介绍了如何通过最简单的单边 N Gram 模型预测单词，还有一种复杂一点的 N Gram 模型通过双边的单词来预测中间的单词，这种模型有个专门的名字，叫 Continuous Bag-of-Words model（CBOW），具体内容差别不大，就不再赘述。

5.4.5 词性判断

上面只使用了词嵌入和 N Gram 模型进行自然语言处理，还没有真正使用循环神经网络，下面介绍 RNN 在自然语言处理中的应用。在这个例子中，我们将使用 LSTM 做词性判断，因为同一个单词有着不同的词性，比如 book 可以表示名词，也可以表示动词，所以需要结合前后文给出具体的判断。先介绍使用 LSTM 做词性判断的原理。

1. 基本原理

定义好一个 LSTM 网络，然后给出一个由很多个词构成的句子，根据前面的内容，每个词可以用一个词向量表示，这样一句话就可以看做是一个序列，序列中的每个元素都是一个高维向量，将这个序列传入 LSTM，可以得到与序列等长的输出，每个输出都表示为对词性的判断，比如名词、动词等。从本质上看，这是一个分类问题，虽然使用了 LSTM，但实际上是根据这个词前面的一些词来对它进行分类，看它是属于几种词性中的哪一种。

思考一下为什么 LSTM 在这个问题里面起着重要的作用。如果完全孤立地对一个词做词性的判断，往往无法得到比较准确的结果，但是通过 LSTM，根据它记忆的特

性，就能够通过这个单词前面记忆的一些词语来对它做一个判断，比如前面的单词如果是 my，那么它紧跟的词很有可能就是一个名词，这样就能够充分地利用上文来处理这个问题。

2. 字符增强

还可以通过引入字符来增强表达，这是什么意思呢？就是说一些单词存在着前缀或者后缀，比如-ly 这种后缀很可能是一个副词，这样我们就能够在字符水平上对词性进行进一步判断，把两种方法集成起来，能够得到一个更好的结果。

在实现上还是用 LSTM，只是这次不再将句子作为一个序列，而是将每个单词作为一个序列。每个单词由不同的字母组成，比如 apple 由 a p p l e 构成，给这些字符建立词向量，形成了一个长度为 5 的序列，将它传入 LSTM 网络，只取最后输出的状态层作为它的一种字符表达，不需要关心提取出来的字符表达到底是什么样，它作为一种抽象的特征，能够更好地预测结果。

5.4.6 词性判断的 PyTorch 实现

作为演示，使用一个简单的训练数据，下面有两句话，每句话中的每个词都给出了词性：

```
training_data = [
    ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
    ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
]
```

接着对单词和词性由 a 到 z 的字符进行编码：

```
word_to_idx = {}
tag_to_idx = {}
for context, tag in training_data:
    for word in context:
        if word not in word_to_idx:
            word_to_idx[word] = len(word_to_idx)
    for label in tag:
        if label not in tag_to_idx:
            tag_to_idx[label] = len(tag_to_idx)

alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

```
character_to_idx = {}
for i in range(len(alphabet)):
    character_to_idx[alphabet[i]] = i
```

接着先定义字符水准上的 LSTM，定义方式和之前类似：

```
class CharLSTM(nn.Module):
    def __init__(self, n_char, char_dim, char_hidden):
        super(CharLSTM, self).__init__()
        self.char_embedding = nn.Embedding(n_char, char_dim)
        self.char_lstm = nn.LSTM(char_dim, char_hidden, batch_first=True)

    def forward(self, x):
        x = self.char_embedding(x)
        _, h = self.char_lstm(x)
        return h[0]
```

定义两层结构：第一层是词嵌入，第二层是 LSTM。在网络的前向传播中，先将单词的 n 个字符传入网络，再通过 nn.Embedding 得到词向量，接着传入 LSTM 网络，得到隐藏状态输出 h，然后通过 h[0] 得到想要的输出状态。对于每个单词，都可以通过 CharLSTM 用相应的字符表示。

接着完成目标，分析每个单词的词性，首先定义好词性的 LSTM 网络：

```
class LSTMTagger(nn.Module):
    def __init__(self, n_word, n_char, char_dim, n_dim, char_hidden,
                 n_hidden, n_tag):
        super(LSTMTagger, self).__init__()
        self.word_embedding = nn.Embedding(n_word, n_dim)
        self.char_lstm = CharLSTM(n_char, char_dim, char_hidden)
        self.lstm = nn.LSTM(n_dim+char_hidden, n_hidden, batch_first=True)
        self.linear1 = nn.Linear(n_hidden, n_tag)

    def forward(self, x, word_data):
        word = [i for i in word_data]
        char = torch.FloatTensor()
        for each in word:
            word_list = []
            for letter in each:
```

```
                word_list.append(character_to_idx[letter.lower()])
            word_list = torch.LongTensor(word_list)
            word_list = word_list.unsqueeze(0)
            tempchar = self.char_lstm(Variable(word_list).cuda())
            tempchar = tempchar.squeeze(0)
            char = torch.cat((char, tempchar.cpu().data), 0)
        char = char.squeeze(1)
        char = Variable(char).cuda()
        x = self.word_embedding(x)
        x = torch.cat((x, char), 1)
        x = x.unsqueeze(0)
        x, _ = self.lstm(x)
        x = x.squeeze(0)
        x = self.linear1(x)
        y = F.log_softmax(x)
        return y
```

看着有点复杂，慢慢来介绍。首先使用 `n_word` 和 `n_dim` 定义单词的词向量矩阵的维度，`n_char` 和 `char_dim` 定义字符的词向量维度，`char_hidden` 表示字符水准上的 LSTM 输出的维度，`n_hidden` 表示每个单词作为序列输入 LSTM 的输出维度，最后 `n_tag` 表示输出的词性分类。介绍完里面参数的含义，下面具体介绍其中网络的向前传播。

学习过 PyTorch 的动态图结构，网络的向前传播就非常简单了。因为要使用字符增强，所以在传入一个句子作为序列的同时，还需要传入句子中的单词，用 `word_data` 表示。动态图结构使得前向传播中可以使用 for 循环将每个单词都传入 CharLSTM，得到的结果和单词的词向量拼在一起作为新的序列输入，将它传入 LSTM 中，最后接一个全连接层，将输出维数定义为词性的数目。

这是基本的思路，就不具体解释每句话的含义了，只是要注意代码里面有 unsqueeze 和 squeeze 的操作，原因前面介绍过，LSTM 的输入要带上 `batch_size`，所以需要将维度扩大。

网络训练经过了 300 次，loss 降到了 0.16 左右。为了验证模型的准确性，可以预测“Everybody ate the apple”这句话中每个词的词性，一共有三种词：DET、NN、V。最后得到的结果如图5.29所示。

结果是一个 4 行 3 列的向量，每一行表示一个单词，每一列表示一种词性，从左到右的词性分别是 DET、NN、V。从每行里面取最大值，那么第一个词的词性就是 NN，

第二个词是 V，第三个词是 DET，第四个词是 NN，与想要的结果相符。

```
Variable containing:
-3.4696 -0.0925 -2.8611
-2.0119 -2.2739 -0.2700
-0.1937 -2.5467 -2.3254
-2.5916 -0.1177 -3.3210
[torch.FloatTensor of size 4x3]
```

图 5.29　网络训练结果

以上，通过几个简单的例子介绍了循环神经网络在自然语言处理中的应用，当然真正的应用会更多，同时也更加复杂，这里就不再深入介绍了，对自然语言处理感兴趣的读者可以进行更深入地探究。

5.5　循环神经网络的更多应用

前面介绍了循环神经网络在时间序列数据和自然语言处理中的简单应用，循环神经网络的应用不止于此，下面简单地介绍现实中循环神经网络更多的应用。

5.5.1　Many to one

循环神经网络不仅能够输入序列、输出序列、还能够输入序列，输出单个向量。只需要在输出的序列里面取其中的一个就可以，通常是取最后一个。这样的结构被称为 Many to one，那么这种结构能够做什么任务呢?

第一个任务是情感分析，将一句话作为序列输入网络，输出只取最后一个，根据输出判断这句话的态度是消极的还是积极的；第二个任务是关键字提取，原理也是一样的，将一句话作为序列输入网络，输出只取最后一个，不同的是用它来表示这句话的关键字。具体的结构可以用图5.30所示。

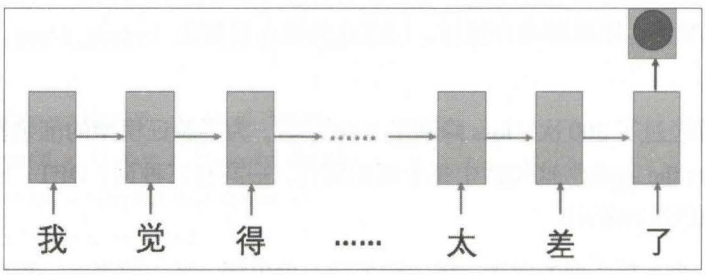


图 5.30　Many to one

5.5.2 Many to Many（shorter）

第二种结构就是输入和输出都是序列，但是输出的序列比输入的序列短。这种类型的结构通常会在语音识别中遇到，因为一段话如果用语音表达往往会比这段话更长，可以看看图 5.31。

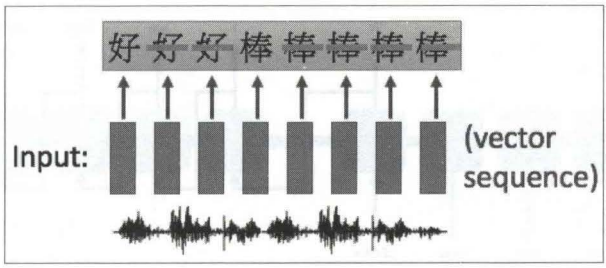


图 5.31 Many to Many

这种情况需要使用 CTC 算法解决重复的问题，CTC 就是将输出的所有可能列举出来，然后通过去重复、去空格的方式来选择最大的概率，这里就不再赘述。

5.5.3 Seq2seq

第三种情况是输出的长度不确定，这种情况一般是在机器翻译的任务中出现，将一句中文翻译成英文，那么这句英文的长度有可能会比中文短，也有可能会比中文长，所以这时候输出的长度就不确定了，需要用序列到序列的模型来解决这个问题，具体如图5.32所示。

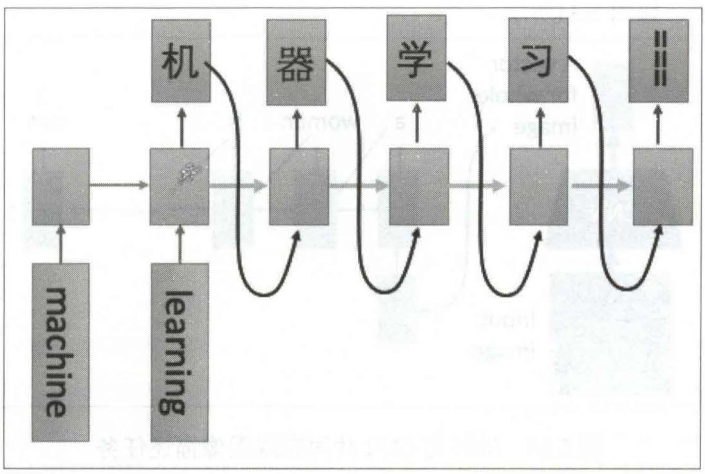


图 5.32 序列到序列模型

在网络结构中，输入一个英文序列，然后输出它对应的中文翻译，输出的部分通过前面的结果预测后面，根据上面的例子，也就是先输出“机”，将它作为下一次的输入，接着输出“器”，这样就能够输出任意长的序列。

聊天机器人和问答系统也都是同样的原理，将句子输入，输出是根据前面的输入来得到，如图 5.33所示。

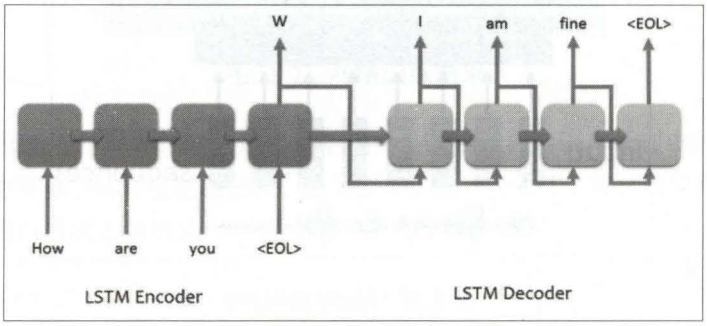


图 5.33　聊天机器人原理

除此之外还可以引入注意力机制强化模型效果，这里就不再赘述，本书的实战部分会详细地介绍机器翻译和注意力机制。

5.5.4　CNN+RNN

RNN 和 CNN 还能够联合在一起完成图像描述任务，简而言之，就是通过预训练的卷积神经网络提取图片特征，接着通过循环网络将特征变成文字描述，具体细节就不再展开介绍了，如图 5.34所示。

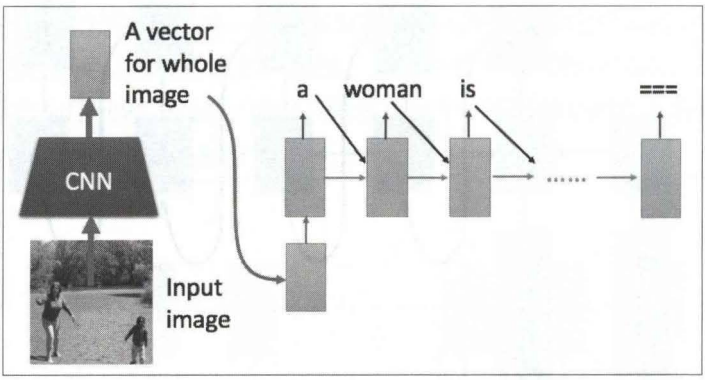


图 5.34　RNN 与 CNN 共同完成图像描述任务

本章首先介绍循环神经网络的基础，其次介绍流行的网络结构：LSTM 和 GRU，接

着用 PyTorch 实现了几个例子，然后介绍循环神经网络是如何应用于自然语言处理的，最后简单介绍循环神经网络在现实中的应用任务。下一章将介绍时下非常流行的生成对抗网络。

第 6 章

生成对抗网络

长久以来，人们都希望机器能够充满创造力，不仅能完成重复的机械劳动，还能完成一些创造性的工作，比如画画、写诗、创作歌词等。这些一直是人工智能长久以来的梦想，随着自动编码器和变分编码器的提出，这一梦想慢慢变成了现实，到了 2014 年，Ian Goodfellow 提出了生成对抗网络（Generative Adversarial Networks, GANs）这一概念，推进了整个无监督学习的发展进程。

这一章将从最简单的生成模型 (Generative Model) 入手，包括自动编码器（Autoencoder）和变分自动编码器（Variational AutoEncoder, VAE），接着详细地介绍生成对抗网络的创新和原理，以及为何生成对抗网络能够成为现在最热门的研究领域。最后介绍生成对抗网络的变式和应用，以及用 PyTorch 实现简单的生成对抗网络。

6.1 生成模型

生成模型（Generative Model）这一概念属于概率统计和机器学习，是指一系列用于随机生成可观测数据的模型。简而言之，就是“生成”的样本和“真实”的样本尽可能地相似。生成模型的两个主要功能就是学习一个概率分布 $P_{model}(X)$ 和生成数据，这是非常重要的，不仅能够用在无监督学习中，还可以用在监督学习中。

前面已经介绍过了适用于图像的卷积神经网络、适用于序列的循环神经网络，但是要知道 Lecun 提出第一代卷积神经网络 Lenet 的时间是 1998 年，而循环神经网络提出的时间更早，是在 1986 年。这些网络在当时并没有流行起来，如今随着计算能力的加强，数据集的增多，深度学习逐渐流行了起来，随着越来越多人的研究，各种各样的

神经网络都在不断进步，CNN 里面出现了 inception net、resnet 等，RNN 演变了 LSTM 和 GRU 等不同的长短时间记忆的网络，虽然神经网络在不断发展，但本质上仍然是在 CNN 和 RNN 的基础上做着监督学习的任务，无监督学习的发展一直比较缓慢，生成模型希望能够让无监督学习取得比较大的进展。

6.1.1 自动编码器

自动编码器（AutoEncoder）最开始作为一种数据的压缩方法，其特点有:

（1）跟数据相关程度很高，这意味着自动编码器只能压缩与训练数据相似的数据，因为使用神经网络提取的特征一般是高度相关于原始的训练集，使用人脸训练出来的自动编码器在压缩自然界动物的图片时表现就会比较差，因为它只学习到了人脸的特征，而没有学习到自然界图片的特征。

（2）压缩后数据是有损的，这是因为在降维的过程中不可避免地要丢失信息。

到了 2012 年，人们发现在卷积神经网络中使用自动编码器做逐层预训练可以训练更深层的网络，但是人们很快发现，良好的初始化策略要比复杂的逐层预训练有效得多，2014 年出现的 Batch Normalization 技术也使得更深的网络能够被有效训练，到了 2015 年年底，通过残差（ResNet）基本可以训练任意深度的神经网络。

所以现在自动编码器主要应用在两个方面：第一是数据去噪，第二是进行可视化降维。自动编码器还有一个功能，即生成数据。

首先给出自动编码器的一般结构，如图 6.1所示。

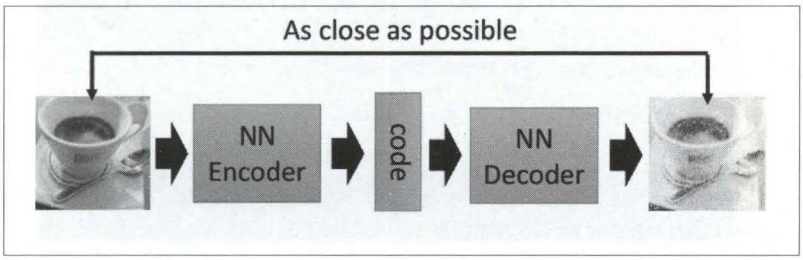


图 6.1 自动编码器结构

从图6.1中能够看到两个部分：第一个部分是编码器（Encoder），第二个部分是解码器（Decoder），编码器和解码器都可以是任意的模型，通常使用神经网络模型作为编码器和解码器。输入的数据经过神经网络降维到一个编码（code），接着又通过另外一个神经网络去解码得到一个与输入原数据一模一样的生成数据，然后通过比较这两个数据，最小化它们之间的差异来训练这个网络中编码器和解码器的参数。当这个过程

训练完之后，拿出这个解码器，随机传入一个编码（code），通过解码器能够生成一个和原数据差不多的数据，图6.2就是希望能够生成一张差不多的图片。

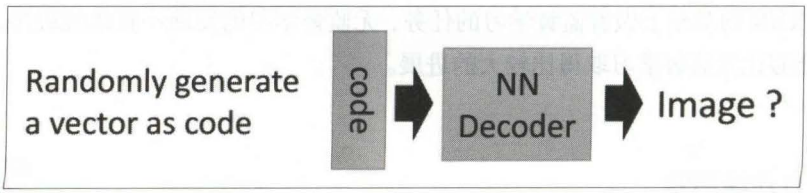


图 6.2　编码与解码

这件事情能不能实现呢？其实是可以的，下面会用 PyTorch 来简单地实现一个自动编码器。

首先构建一个简单的多层全连接网络。

对 MNIST 数据里面的大小做一下变换，使用 `transforms.Normalize()` 使得图片的大小变为-1 ～ 1 之间，这是为了输入变成一个比较对称的分布，训练更加容易收敛。接着通过 DataLoader 生成一个读取图片的迭代器。

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.ReLU(True),
            nn.Linear(128, 64),
            nn.ReLU(True),
            nn.Linear(64, 12),
            nn.ReLU(True),
            nn.Linear(12, 3)
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 128),
            nn.ReLU(True),
            nn.Linear(128, 28*28),
            nn.Tanh()
```

```
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

定义一个简单的 4 层网络作为编码器，中间使用 ReLU 激活函数，最后输出的维度是三维的，定义的解码器，输入三维的编码，输出一个 28 × 28 的图像数据，特别要注意最后使用的激活函数是 Tanh，这个激活函数能够将最后的输出转换到-1 ～ 1 之间，这是因为输入的图片已经变换到了-1 ～ 1 之间，这里的输出必须和它对应。

在开始训练网络的时候，每次编码器的输入要展开乘 28 × 28 = 784 的大小，然后输入编码器得到编码，再输入解码器得到生成的图片，然后和原图片去比较，通过 `save_image()` 这个函数输出网络得到的结果。

看看自动编码器的数据结果到底是怎么样的，如图 6.3所示。

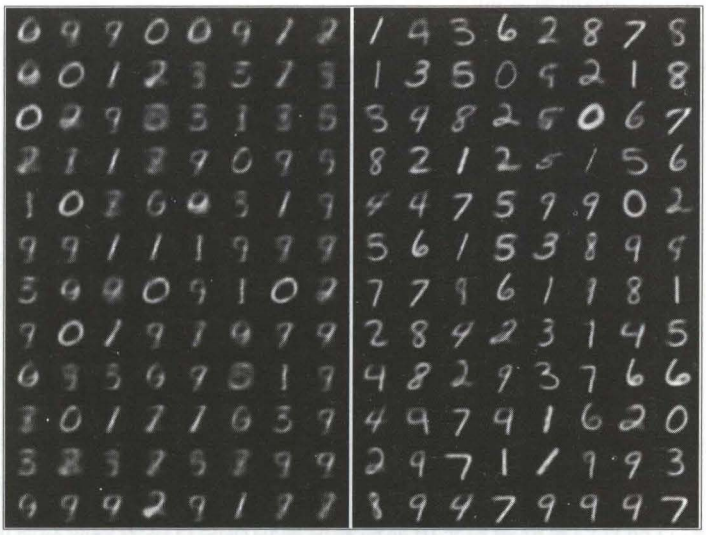


图 6.3 数据结果

图 6.3 左边是刚训练 20 次得到的效果，图片还比较模糊。图 6.3 右边是 40 次之后的效果，可以看到图片已经有点清晰了，但轮廓还是有些模糊，所以说多层感知器能够做得比较有限，生成的图片还是比较模糊的。

可以将编码的分布可视化出来，具体看看随机给一个三维的编码，能够生成的图片的分布，如图6.4与图6.5所示。

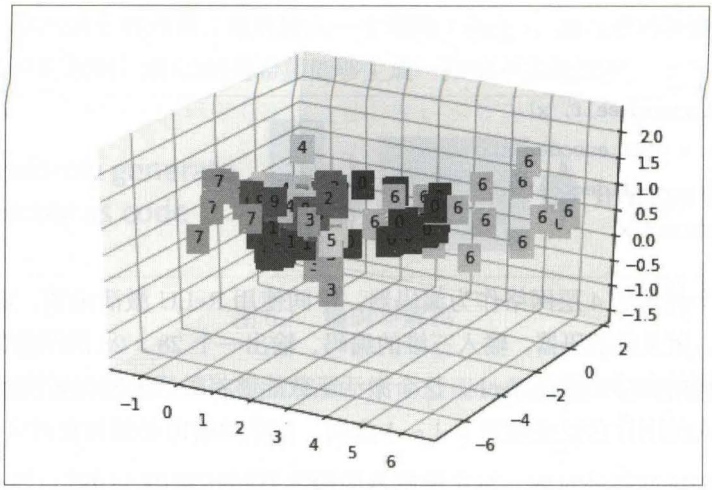


图 6.4　三维分布

图 6.4 是一个三维的分布，图 6.5 是一个二维的分布，可以看出生成的图片是如何根据所给出的随机编码而改变的。

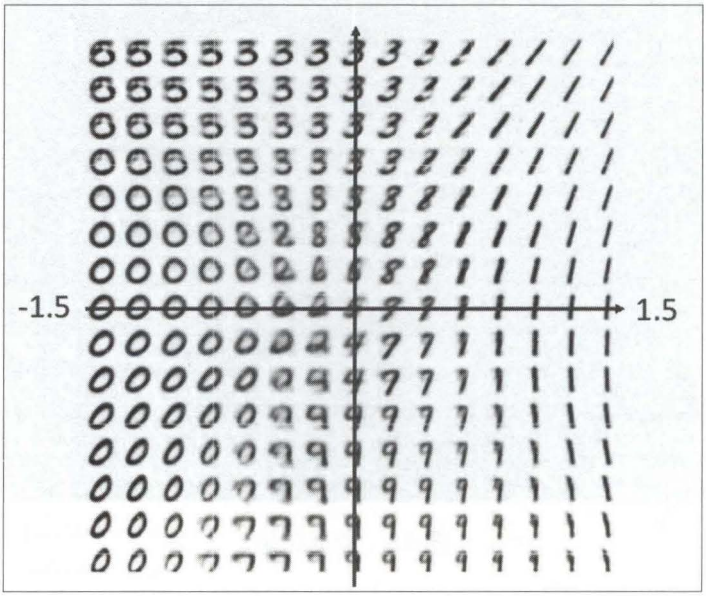


图 6.5　二维分布

接下来希望使用一个更加复杂的网络做自动编码器，这样能够使生成的图片效果更好，而对于图片而言，最好的图像处理网络就是卷积神经网络，所以可以构建一个卷积神经网络做自动编码器。

```
class DCautoencoder(nn.Module):
    def __init__(self):
        super(DCautoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=3, padding=1),  # b, 16, 10, 10
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=2),  # b, 16, 5, 5
            nn.Conv2d(16, 8, 3, stride=2, padding=1),  # b, 8, 3, 3
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=1)  # b, 8, 2, 2
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),  # b, 16, 5, 5
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1),  # b, 8, 15,
            15
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),  # b, 1, 28,
            28
            nn.Tanh()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

这里的编码器使用了多层卷积神经网络，解码器使用了 `nn.ConvTranspose2d()`，看作卷积的反操作，可以在某种意义上看成是反卷积。

使用卷积神经网络作为自动编码器和前面使用多层感知器定义的自动编码器一样，最后得到的结果会比简单的多层感知器好，可以看如图 6.6所示的结果对比。

图6.6左边是多层全连接神经网络的结果，图6.6右边是卷积神经网络的结果，可以看出其实两者之间的差别不大，只不过多层感知器的结果会稍微模糊一些。

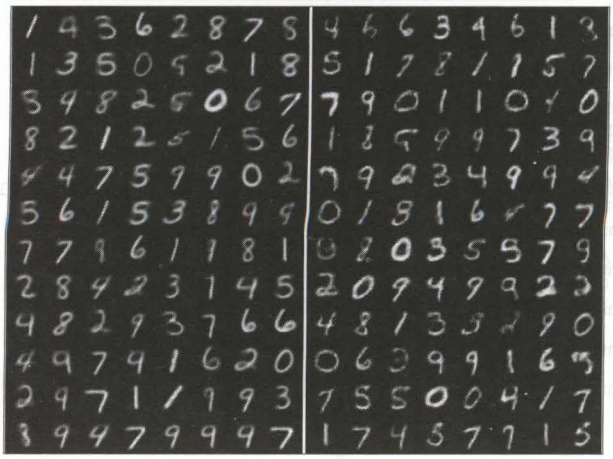


图 6.6　结果对比

6.1.2　变分自动编码器

变分自动编码器（Variational AutoEncoder）是自动编码器的升级版本，它的结构跟自动编码器是相似的，也是由编码器和解码器构成的。

在自动编码器中，需要输入一张图片，然后将一张图片编码之后得到一个隐含向量，这比随机取一个随机向量好，因为这包含着原图片的信息，然后将隐含向量解码得到与原图片对应的照片。

但是这样其实并不能任意生成图片，因为没办法自己去构造隐藏向量，需要通过一张图片输入编码才知道得到的隐含向量是什么，这时就可以通过变分自动编码器解决这个问题。

其实原理特别简单，只需要在编码过程给它增加一些限制，迫使它生成的隐含向量能够粗略地遵循一个标准正态分布，这就是它与一般的自动编码器最大的不同。

这样生成一张新图片就很简单了，只需要给它一个标准正态分布的随机隐含向量，通过解码器就能够生成想要的图片，而不需要先给它一张原始图片编码。

在实际情况中，需要在模型的准确率上与隐含向量服从标准正态分布之间做一个权衡，所谓模型的准确率就是指解码器生成的图片与原图片的相似程度。可以让神经网络自己做这个决定，只需要将这两者都做一个 loss，然后再将它们求和作为总的 loss，这样网络就能够自己选择如何做才能使得这个总的 loss 下降。另外要衡量两种分布的相似程度，需要引入一个新的概念，KL divergence，这是用来衡量两种分布相似程度的统计量，它越小，表示两种概率分布越接近。

对于离散的概率分布，定义如下：

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \tag{6.1}$$

对于连续的概率分布，定义如下：

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{6.2}$$

这里就是用 KL divergence 表示隐含向量与标准正态分布之间差异的 loss，另外一个 loss 仍然使用生成图片与原图片的均方误差来表示。

这里变分编码器使用了一个技巧——“重新参数化”来解决 KL divergence 的计算问题。

这时不再是每次生成一个隐含向量，而是生成两个向量：一个表示均值，一个表示标准差，然后通过这两个统计量合成隐含向量，用一个标准正态分布先乘标准差再加上均值就行了，这里默认编码之后的隐含向量是服从一个正态分布的。这个时候要让均值尽可能接近 0，标准差尽可能接近 1。论文里面有详细的推导介绍如何得到这个 loss 的计算公式，有兴趣的读者可以去 https://arxiv.org/pdf/1606.05908.pdf 看看推导过程。

这个问题可以使用 PyTorch 轻松地实现：

```
reconstruction_function = nn.BCELoss(size_average=False)  # mse loss

def loss_function(recon_x, x, mu, logvar):
    """
    recon_x: generating images
    x: origin images
    mu: latent mean
    logvar: latent log variance
    """
    BCE = reconstruction_function(recon_x, x)
    # loss = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)
    # KL divergence
```

```
    return BCE + KLD
```

另外变分编码器除了可以随机生成隐含变量，还能够提高网络的泛化能力。

最后是 VAE 网络定义的代码实现：

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        if torch.cuda.is_available():
            eps = torch.cuda.FloatTensor(std.size()).normal_()
        else:
            eps = torch.FloatTensor(std.size()).normal_()
        eps = Variable(eps)
        return eps.mul(std).add_(mu)

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return F.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparametrize(mu, logvar)
        return self.decode(z), mu, logvar
```

VAE 的结果比普通的自动编码器要好很多，可以将它和卷积神经网络做的标准自动编码器得到的结果进行比较。

图 6.7 左边是标准自动编码器，右边是变分编码器，可以看到右边的结果明显要比左边的结果清晰。

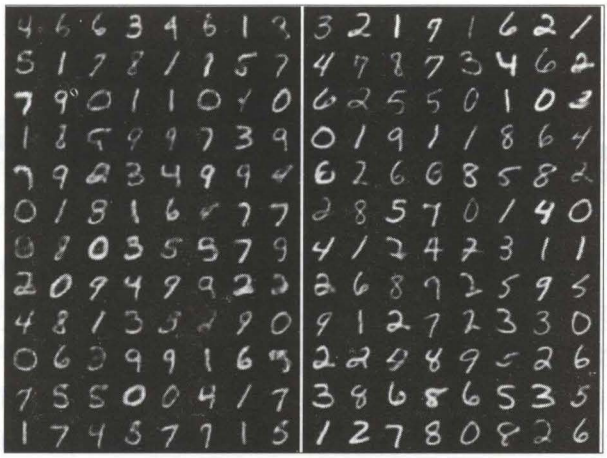


图 6.7 VAE 与标准自动编码器结果对比

虽然变分编码器有这些优点，但是它同样有自动编码器的缺点，那就是直接计算生成图片和原始图片的均方误差来作为损失函数，使得生成的图片会有点模糊。

接下来介绍生成对抗网络是如何解决这个问题的。

6.2 生成对抗网络

2014 年，深度学习三巨头之一 Ian Goodfellow 提出了生成对抗网络（Generative Adversarial Networks, GANs）这一概念，刚开始并没有引起轰动，直到 2016 年，学界、业界对它的兴趣如“井喷”一样爆发，多篇重磅文章陆续发表，Lecun 这样形容 GANs “adversarial training is the coolest thing since sliced bread”。2016 年 12 月 NIPS 大会上，Goodfellow 做了关于 GANs 的专题报告，使得 GANs 成为了当今最热门的研究领域之一，接下来介绍如今深度学习界的明星——生成对抗网络。

6.2.1 何为生成对抗网络

生成对抗网络，根据它的名字，可以推断这个网络由两部分组成：第一部分是生成，第二部分是对抗。这个网络的第一部分是生成模型，就像之前介绍的自动编码器的解码部分；第二部分是对抗模型，严格来说它是一个判断真假图片的判别器。生成对抗网络最大的创新在此，这也是生成对抗网络与自动编码器最大的区别。简单来说，生成

对抗网络就是让两个网络相互竞争，通过生成网络来生成假的数据，对抗网络通过判别器判别真伪，最后希望生成网络生成的数据能够以假乱真骗过判别器。

过程如图 6.8所示。

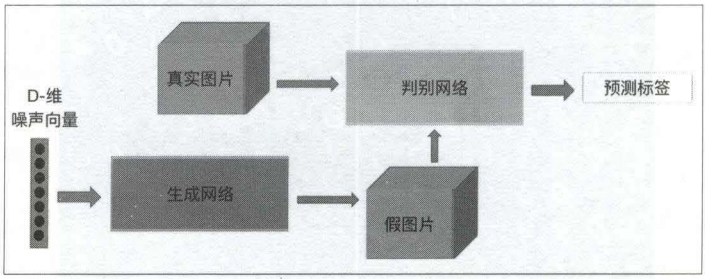


图 6.8　生成对抗网络生成数据过程

下面依次介绍生成模型和对抗模型。

1. 生成模型

首先看看生成模型，前一节自动编码器其实已经给出了一般的生成模型。

在生成对抗网络中，不再是将图片输入编码器得到隐含向量然后生成图片，而是随机初始化一个隐含向量，根据变分自动编码器的特点，初始化一个正态分布的隐含向量，通过类似解码的过程，将它映射到一个更高的维度，最后生成一个与输入数据相似的数据，这就是假的图片。这时自动编码器是通过对比两张图片之间每个像素点的差异计算损失函数的，而生成对抗网络会通过对抗过程来计算出这个损失函数，如图6.9所示。

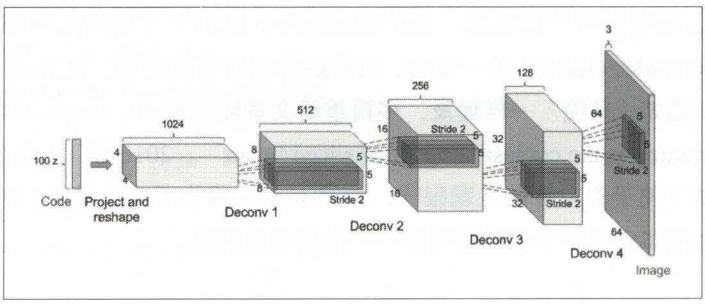


图 6.9　生成模型

2. 对抗模型

重点来介绍对抗过程，这个过程是生成对抗网络相对于之前的生成模型如自动编码器等最大的创新。

对抗过程简单来说就是一个判断真假的判别器，相当于一个二分类问题，输入一张

真的图片希望判别器输出的结果是 1，输入一张假的图片希望判别器输出的结果是 0。这跟原图片的 label 没有关系，不管原图片到底是一个多少类别的图片，它们都统一称为真的图片，输出的 label 是 1，则表示是真实的；而生成图片的 label 是 0，则表示是假的。

在训练的时候，先训练判别器，将假的数据和真的数据都输入给判别模型，这个时候优化这个判别模型，希望它能够正确地判断出真的数据和假的数据，这样就能够得到一个比较好的判别器。

然后开始训练生成器，希望它生成的假的数据能够骗过现在这个比较好的判别器。具体做法就是将判别器的参数固定，通过反向传播优化生成器的参数，希望生成器得到的数据在经过判别器之后得到的结果能尽可能地接近 1，这时只需要调整一下损失函数就可以了，之前在优化判别器的时候损失函数是让假的数据尽可能接近 0，而现在训练生成器的损失函数是让假的数据尽可能接近 1。

这其实就是一个简单的二分类问题，这个问题可以用前面介绍过的很多方法去处理，比如 Logistic 回归、多层感知器、卷积神经网络、循环神经网络等。

上面是生成对抗网络的简单解释，可以通过代码更清晰地展示整个过程。

跟自动编码器一样，先使用简单的多层感知器来实现：

```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()
        self.dis = nn.Sequential(
            nn.Linear(784, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.dis(x)
        return x
```

上面是判别器的结构，中间使用了斜率设为 0.2 的 LeakyReLU 激活函数，最后需要使用 `nn.Sigmoid()` 将结果映射到 0 ～ 1 之间概率进行真假的二分类。这里之所以

用 LeakyReLU 激活函数而不使用 ReLU 激活函数，是因为经过实验，LeakyReLU 的表现更好。

```
class generator(nn.Module):
    def __init__(self,  input_size):
        super(generator, self).__init__()
        self.gen = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(True),
            nn.Linear(256, 256),
            nn.ReLU(True),
            nn.Linear(256, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.gen(x)
        return x
```

这就是生成器的结构，跟自动编码器中的解码器是类似的，最后需要使用 `nn.Tanh()`，将数据分布到 $-1 \sim 1$ 之间，这是因为输入的图片会规范化到 $-1 \sim 1$ 之间。

接着需要定义损失函数和优化函数：

```
criterion = nn.BCELoss()  # Binary Cross Entropy
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)
```

这里使用二分类的损失函数 `nn.BCELoss()`，使用 Adam 优化函数，学习率设置为 0.0003。

接着是最为重要的训练过程，这个过程分为两个部分：一个是判别器的训练，一个是生成器的训练。

首先来看看判别器的训练。

```
img = img.view(num_img, -1)
real_img = Variable(img).cuda()
real_label = Variable(torch.ones(num_img)).cuda()
fake_label = Variable(torch.zeros(num_img)).cuda()

```

```
# compute loss of real_img
real_out = D(real_img)
d_loss_real = criterion(real_out, real_label)
real_scores = real_out

# compute loss of fake_img
z = Variable(torch.randn(num_img, z_dimension)).cuda()
fake_img = G(z)
fake_out = D(fake_img)
d_loss_fake = criterion(fake_out, fake_label)
fake_scores = fake_out

# bp and optimize
d_loss = d_loss_real + d_loss_fake
d_optimizer.zero_grad()
d_loss.backward()
d_optimizer.step()
```

开始需要自己创建 label，真实的数据是 1，生成的假的数据是 0，然后将真实的数据输入判别器得到 loss，将假的数据输入判别器得到 loss，将这两个 loss 加起来得到总的 loss，然后反向传播去更新参数就能够得到一个优化好的判别器。

接下来是生成模型的训练：

```
# compute loss of fake_img
z = Variable(torch.randn(num_img, z_dimension)).cuda()  # 得到随机噪声
fake_img = G(z)  # 生成假的图片
output = D(fake_img)  # 经过判别器得到结果
g_loss = criterion(output, real_label)  # 得到假的图片与真实图片label的loss

# bp and optimize
g_optimizer.zero_grad()  # 归0梯度
g_loss.backward()  # 反向传播
g_optimizer.step()  # 更新生成网络的参数
```

一个随机隐含向量通过生成网络得到了一个假的数据，然后希望假的数据经过判别模型后尽可能和真实 label 接近，通过 `g_loss = criterion(output, real_label)` 实现，然后反向传播去优化生成器的参数，在这个过程中，判别器的参数不再发生变化，否则生成器永远无法骗过优化的判别器。

除了使用简单的多层感知器外，也可以在生成模型和对抗模型中使用更加复杂的卷积神经网络，定义十分简单。

```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, 5, padding=2),  # batch, 32, 28, 28
            nn.LeakyReLU(0.2, True),
            nn.AvgPool2d(2, stride=2),  # batch, 32, 14, 14
            )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, 5, padding=2),  # batch, 64, 14, 14
            nn.LeakyReLU(0.2, True),
            nn.AvgPool2d(2, stride=2)  # batch, 64, 7, 7
        )
        self.fc = nn.Sequential(
            nn.Linear(64*7*7, 1024),
            nn.LeakyReLU(0.2, True),
            nn.Linear(1024, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        '''
        x: batch, width, height, channel=1
        '''
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

class generator(nn.Module):
    def __init__(self, input_size, num_feature):
        super(generator, self).__init__()
        self.fc = nn.Linear(input_size, num_feature)  # batch, 3136=1x56x56
```

```
        self.br = nn.Sequential(
            nn.BatchNorm2d(1),
            nn.ReLU(True)
        )
        self.downsample1 = nn.Sequential(
            nn.Conv2d(1, 50, 3, stride=1, padding=1),  # batch, 50, 56, 56
            nn.BatchNorm2d(50),
            nn.ReLU(True)
        )
        self.downsample2 = nn.Sequential(
            nn.Conv2d(50, 25, 3, stride=1, padding=1),  # batch, 25, 56, 56
            nn.BatchNorm2d(25),
            nn.ReLU(True)
        )
        self.downsample3 = nn.Sequential(
            nn.Conv2d(25, 1, 2, stride=2),  # batch, 1, 28, 28
            nn.Tanh()
        )

    def forward(self, x):
        x = self.fc(x)
        x = x.view(x.size(0), 1, 56, 56)
        x = self.br(x)
        x = self.downsample1(x)
        x = self.downsample2(x)
        x = self.downsample3(x)
        return x
```

图 6.10 左边是多层感知器的生成对抗网络，右边是卷积生成对抗网络，右边的图片比左边的图片噪声明显更少。在卷积神经网络里引入了批标准化（Batchnormalization）来稳定训练，同时使用了 LeakyReLU 和平均池化来进行训练。生成对抗网络的训练其实是很困难的，因为这是两个对偶网络在相互学习，所以需要增加一些训练技巧才能使训练更加稳定。

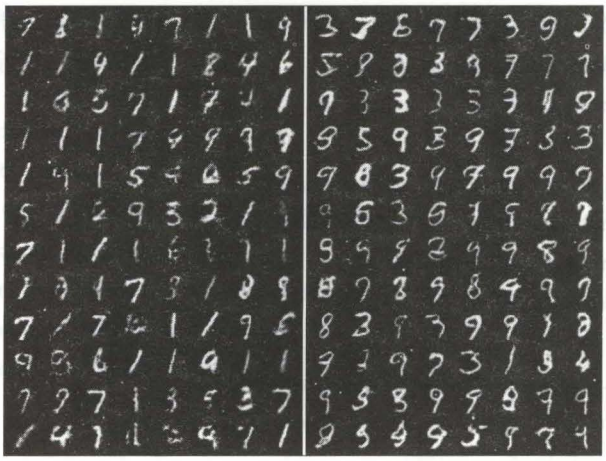


图 6.10　生成对抗网络对比结果

以上介绍了生成对抗网络的简单原理和训练流程，但是对生成对抗网络而言，它其实并没有真正地学习到它要表示的物体，通过对抗的过程，它只是生成了一张尽可能真的图片，这就意味着没办法决定用哪种噪声能够生成想要的图片，除非把初始分布都试一遍。所以在生成对抗网络提出之后，有很多基于标准生成对抗网络的变式来解决各种各样的问题，我们会在 6.3 节介绍。

6.2.2　生成对抗网络的数学原理

上一节介绍了什么是生成对抗，这一节将会用严格的数学语言证明生成对抗网络的合理性。

首先需要一点预备知识，KL divergence，这是统计中的一个概念，用于衡量两种概率分布的相似程度，数值越小，表示两种概率分布越接近。离散的概率分布，定义如下：

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \tag{6.3}$$

连续的概率分布，定义如下：

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{6.4}$$

根据之前介绍的内容，要做的操作如图 6.11所示。

想要将一个随机高斯噪声 z 通过一个生成网络 G 得到一个和真的数据分布 $p_{data}(x)$

差不多的生成分布 $p_G(x;\theta)$，其中参数 θ 是网络的参数决定的，希望找到 θ 使得 $P_G(x;\theta)$ 和 $P_{data}(x)$ 尽可能接近。

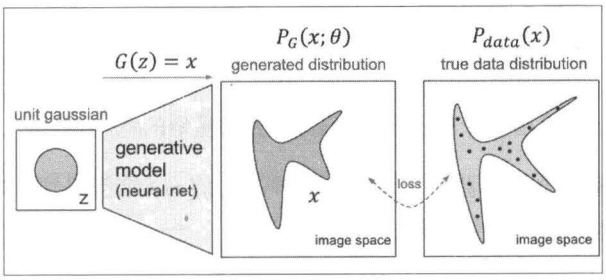


图 6.11 操作过程

从真实数据分布 $P_{data}(x)$ 里面取样 m 个点，$\{x^1, x^2, \cdots, x^m\}$，根据给定的参数 θ 可以计算如下的概率 $P_G(x^i;\theta)$，那么生成 m 个样本数据的似然（likelihood）就是：

$$L = \prod_{i=1}^{m} P_G(x^i;\theta) \tag{6.5}$$

找到 θ^* 来最大化这个似然估计：

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{m} p_G(x^i;\theta) \Leftrightarrow \arg\max_{\theta} \log \prod_{i=1}^{m} P_G(x^i;\theta) \tag{6.6}$$

$$= \arg\max_{\theta} \sum_{i}^{m} \log P_G(x^i;\theta) \tag{6.7}$$

$$\approx \arg\max_{\theta} E_{x\sim P_{data}}[\log P_G(x;\theta)] \tag{6.8}$$

$$\Leftrightarrow \arg\max_{\theta} \int_x P_{data}(x) \log P_G(x;\theta) dx - \int_x P_{data}(x) \log P_{data}(x) dx \tag{6.9}$$

$$= \arg\max_{\theta} \int_x P_{data}(x) \log \frac{P_G(x;\theta)}{P_{data}(x)} dx \tag{6.10}$$

$$= \arg\min_{\theta} KL(P_{data}(x)||P_G(x;\theta)) \tag{6.11}$$

$$\tag{6.12}$$

而 $P_G(x;\theta)$ 如何算出来呢？

$$P_G(x) = \int_z P_{prior}(z) I_{[G(z)=x]} dz \tag{6.13}$$

里面的 I 表示示性函数，也就是：

$$I_{G(z)=x} = \begin{cases} 0 & G(z) \neq x \\ 1 & G(z) = x \end{cases} \tag{6.14}$$

这样其实根本没办法求出这个 $P_G(x)$，这就是生成模型的基本想法。

Generator G 是一个生成器，给定先验分布 $P_{prior}(z)$，希望得到生成分布 $P_G(x)$，这里很难通过极大似然估计得到结果。

Discriminator D 中 D 是一个函数，用来衡量 $P_G(x)$ 与 $P_{data}(x)$ 之间的差距，可用来取代极大似然估计。

首先定义函数 $V(G,D)$ 如下：

$$V(G,D) = E_{x\sim P_{data}}[\log D(X)] + E_{x\sim P_G}[\log(1-D(X))] \tag{6.15}$$

可以通过下面的式子求得最优的生成模型：

$$G^* = \arg\min_G \max_D V(G,D) \tag{6.16}$$

为什么定义了一个 $V(G,D)$ 然后通过求 max 和 min 就能够取得最优的生成模型呢？

首先我们只考虑 $\max_D V(G,D)$，看它表示什么含义。

在给定 G 的前提下，取一个合适的 D 使得 $V(G,D)$ 能够取得最大值，这就是简单的微积分。

$$V = E_{x\sim P_{data}}[\log D(X)] + E_{x\sim P_G}[\log(1-D(x))] \tag{6.17}$$

$$= \int_x P_{data}(x)\log D(x)dx + \int_x p_G(x)\log(1-D(x))dx \tag{6.18}$$

$$= \int_x [P_{data}(x)\log D(X)] + P_G(x)\log(1-D(x))]dx \tag{6.19}$$

对于这个积分，要取它的最大值，希望对于给定的 x，积分里面的项是最大的，也就是希望取到一个最优的 D^* 最大化下面这个式子：

$$P_{data}(x)\log D(x) + P_G(x)\log(1-D(x)) \tag{6.20}$$

在数据给定、G 给定的前提下，$P_{data}(x)$与$P_G(x)$ 都可以看作常数，分别用 a，b 来表示

它们，这样就可以得到如下的式子：

$$f(D) = a\log(D) + b\log(1-D) \tag{6.21}$$

$$\frac{df(D)}{dD} = a \times \frac{1}{D} + b \times \frac{1}{1-D} \times (-1) = 0 \tag{6.22}$$

$$a \times \frac{1}{D^*} = b \times \frac{1}{1-D^*} \tag{6.23}$$

$$\Leftrightarrow a \times (1-D^*) = b \times D^* \tag{6.24}$$

$$D^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_G(x)} \tag{6.25}$$

这样就求得了在给定 G 的前提下，能够使得 $V(D)$ 取得最大值的 D，将 D 代回原来的 $V(G,D)$，得到如下的结果：

$$\max V(G,D) = V(G,D^*) \tag{6.26}$$

$$= E_{x\sim P_{data}}[\log \frac{P_{data}(x)}{P_{data}(x) + P_G(x)}] + E_{x\sim P_G}[\log \frac{P_G(x)}{P_{data}(x) + P_G(x)}] \tag{6.27}$$

$$= \int_x P_{data}(x) \log \frac{\frac{1}{2}P_{data}}{\frac{P_{data}(x)+P_G(x)}{2}} dx + \int_x P_G(x) \log \frac{\frac{1}{2}P_G(x)}{\frac{P_{data}(x)+P_G(x)}{2}} dx \tag{6.28}$$

$$= -2\log 2 + KL(P_{data}(x)||\frac{P_{data}(x) + P_G(x)}{2}) + KL(P_G(x)||\frac{P_{data}(x) + P_G(x)}{2}) \tag{6.29}$$

$$= -2\log 2 + 2JSD(P_{data}(x)||P_G(x)) \tag{6.30}$$

这里引入了一个新的概念，JS Divergence，定义如下：

$$JSD(P||Q) = \frac{1}{2}D(P||M) + \frac{1}{2}D(Q||M) \quad M = \frac{1}{2}(P+Q) \tag{6.31}$$

通过上面的定义知道 KL Divergence 其实是不对称的，而 JS Divergence 是对称的，它们都能够用于衡量两种分布之间的差异。看到这里其实就已经推导出了为什么这么衡量是有意义的，因为取 D 使得 $V(G,D)$ 取得 max 值，这个时候这个 max 值是由两个 KL divergence 构成的，相当于这个 max 的值就是衡量 $P_G(x)$与$P_{data}(x)$ 的差异程度，所以这个时候取：

$$\arg(\min_G \max_D V(G,D)) \tag{6.32}$$

就能够取到 G 使得这两种分布的差异性最小，这样自然就能够生成一个和原分布

尽可能接近的分布，同时也摆脱了计算极大似然估计，所以 GAN 的本质是通过改变训练的过程来避免烦琐的计算。

以上详细地介绍了生成对抗网络的数学推导，略微有些烦琐，但是有助于理解生成对抗网络的本质。

6.3 Improving GAN

这一节将介绍改善的生成对抗网络，因为生成对抗网络存在很多问题，所以人们研究能否通过改善网络结构或者损害函数来解决这些问题。

6.3.1 Wasserstein GAN

Wasserstein GAN 是 GAN 的一种变式，我们知道 GAN 的训练是非常麻烦的，需要很多训练技巧，而且在不同的数据集上，由于数据的分布会发生变化，也需要重新调整参数，不仅需要小心地平衡生成器和判别器的训练进程，同时生成的样本还缺乏多样性。除此之外最大的问题是没办法衡量这个生成器到底好不好，因为没办法通过判别器的 loss 去判断这个事情。虽然 DC GAN 依靠对生成器和判别器的结构进行枚举，最终找到了一个比较好的网络设置，但还是没有从根本上解决训练的问题。

WGAN 的出现，彻底解决了下面这些难点：

（1）彻底解决了训练不稳定的问题，不再需要设计参数去平衡判别器和生成器；

（2）基本解决了 collapse mode 的问题，确保了生成样本的多样性；

（3）训练中有一个向交叉熵、准确率的数值指标来衡量训练的进程，数值越小代表 GAN 训练得越好，同时也就代表着生成的图片质量越高；

（4）不需要精心设计网络结构，用简单的多层感知器就能够取得比较好的效果。

下面先介绍为什么 GAN 会有这些缺点，然后解释 WGAN 是通过什么办法解决这些问题的。

1.GAN 的局限性

根据之前介绍的，有下面的式子：

$$\max V(G,D) = V(G,D^*) = -2\log 2 + 2JSD(P_{data}(x)||P_G(x)) \tag{6.33}$$

从式（6.33）我们知道原始的 GAN 是通过最优判别器下的 JS Divergence 来衡量两种分布之间的差异的，而且最优判别器下 JS Divergence 越小，就说明两种分布越接近，但是 JS Divergence 有一个严重的问题，那就是如果两种分布完全没有重叠部分，或者说重叠部分可忽略，那么 JS Divergence 将恒等于常数 log2。换句话说，就算两种分布很接近，但是只要它们没有重叠，那么 JS Divergence 就是一个常数，这就使得网络没办法通过这个损失函数去学习，因为它没办法知道它是否做得好，这就会导致梯度消失，同时这也使得我们没有办法衡量这两种分布到底有多靠近。

而真实分布与生成的分布没有重叠部分的概率有多大呢？其实是非常大的，直观来讲，真实分布是一个高维分布，而生成的分布来自于一个低维分布，所以其实很有可能生成分布和真实分布之间就没有重叠的部分。除此之外，不可能真正去计算两个分布，只能近似取样，所以也导致了两种分布没有重叠部分。如果判别器训练得太好，那么生成的分布和原来分布基本没有重叠部分，这就导致了梯度消失；如果判别器训练得不好，这样生成器的梯度又不准，就会出现错误的优化方向。如果要使得 GAN 能够完美地收敛，那么需要判别器的训练不好也不坏，而这个度是很难把握的，况且这还依赖数据的分布等条件，所以 GAN 才这么难训练。

2.Wasserstein 距离

既然 GAN 存在的问题都是由于 JS Divergence 引起的，那么能不能换一种度量方式去衡量两种分布之间的差异，而不使用 JS Divergence？答案是肯定的，这就是 WGAN 中提出的解决办法。

首先介绍一种新的度量方式去度量两种分布之间的差异——Wasserstein 距离，也称为 Earth Mover 距离，定义如下：

$$W(P_r, P_g) = \inf_{\gamma \sim \Pi(P_r, P_g)} E_{x,y}[||x - y||] \tag{6.34}$$

看上去可能比较复杂，数学解释如下：对于两种分布 P_r 和 P_g，它们的联合分布是 $\Pi(P_r, P_g)$，换句话说 $\Pi(P_r, P_g)$ 中每一个联合分布的边缘分布就是 P_r 或者 P_g。那么对每一个联合分布而言，从里面取样 x 和 y，并计算 x 和 y 的距离，然后取遍所有的 x 和 y 计算一下期望，接着取这些期望里面最小的作为 W 距离的定义。

如果上面的解释不够清楚，也可以通俗地解释，因为它还有一个别名叫 Earth mover 距离，也就是推土机距离，这是什么意思呢？可以把两种分布想象成两堆土，然后想想如何用推土机将一种分布变成另外一种分布的样子，会有很多种移动方案，里面最小消耗的那种方案就是最优的方案，也就是这个距离的定义。

W 距离与 JS Divergence 相比有什么好处呢？最大的好处就是不管两种分布是否

有重叠，它都是连续变换的而不是突变的，可以用下面这个例子来说明一下，如图6.12所示。

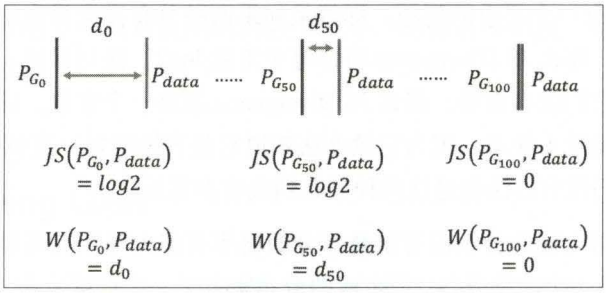


图 6.12　W 距离例子

通过上面这个演示可以发现，虽然两种分布更接近，但 JS Divergence 仍然是 log2，W 距离就能够连续而有效地衡量两种分布之间的差异。

3.WGAN

W 距离有很好的优越性，把它拿来作为两种分布的度量优化生成器，但是 W 距离里面有一个 $\inf_{\gamma\sim\Pi(P_r,P_g)}$ 是没办法求解的。作者 Martin 在论文附录里面通过定理将这个问题转变成了一个新的问题，有着如下形式：

$$W(P_r, P_g) = \frac{1}{K} \sup_{||f||_L \leq K} E_{x\sim P_r}[f(x)] - E_{x\sim P_g}[f(x)] \tag{6.35}$$

这里引入了一个新的概念——Lipschitz 连续。如果函数 f 满足 Lipschitz 连续条件，那么它就满足下面的式子：

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \tag{6.36}$$

我们不希望函数的变化太快，希望函数 f 变化能比较平缓。

那么可以将上面的式子改成 GAN：

$$W(P_r, P_g) \approx \max_{\omega:|D_w|_L \leq K} E_{x\sim P_r}[D(X)] - E_{x\sim P_g}[D(x)] \tag{6.37}$$

也就是说构建一个神经网络 D 作为判别器，希望 D 输出的变化比较平缓，在实际计算中限制 D 中的参数大小不超过某个范围，这样就使得关于输入的样本，D 的输出变化基本不会超过某个范围，所以就能够基本满足 Lipschitz 连续条件。

所以最后构造一个判别器 D，满足：

$$L = E_{x\sim P_r}[D(x)] - E_{x\sim P_g}[D(x)] \tag{6.38}$$

尽可能取到最大，同时 D 还要满足 Lipschitz 连续条件，得到的 L 可以近似为真实分布和生成分布的 Wasserstein 距离。原始的 GAN 做的是二分类的任务，也就是对于真假图片进行二分类，而 WGAN 做的是回归问题，相当于近似拟合 Wasserstein 距离。

最后优化生成器的时候希望最小化 L，这时候需要满足 Lipschitz 连续条件，所以需要做权重的裁剪，由于 W 距离的优越性，不再需要担心梯度消失的问题，这样就能够得到 WGAN 的整个训练过程。

总结一下，WGAN 与原始 GAN 相比，只改了以下四点：

（1）判别器最后一层去掉 sigmoid；

（2）生成器和判别器的 loss 不取 log；

（3）每次更新判别器的参数之后把它们的绝对值裁剪到不超过一个固定常数的数；

（4）不要用基于动量的优化算法（比如 momentuem 和 Adam），推荐使用 RMSProp。

前三点都是从理论分析得到的结果，第（4）点是作者从实验中发现的。对于 WGAN，论文作者做了不少实验，得到了几个结论：第一，WGAN 如果使用类似 DCGAN 的结构，那么和 DCGAN 生成的图片差不多，但是 WGAN 的优势就在于不用 DCGAN 的结构，也能生成效果比较好的图片，但是把 DCGAN 的 Batch Normalization 拿掉的话，DCGAN 就不能生成图片了；第二，WGAN 和原始的 GAN 都是用多层全连接网络的话，WGAN 生成的图片质量会变得差一些，但是原始的 GAN 不仅质量很差，还有多样性不足的问题。

6.3.2 Improving WGAN

WGAN 的提出成功地解决了 GAN 的很多问题，最后需要满足一阶 Lipschitz 连续性条件，所以在训练的时候加了一个限制——权重裁剪。

然而权重的裁剪只是一种简单的做法，不是最好的做法，所以随后有人提出了一些新的办法来解决这个问题。

首先提出一个定理：一个可微函数如果满足 1 阶 Lipschitz 连续，等价于它的梯度范数处小于 1。用式子来表示就是：

$$||D||_L \leq 1 \Leftrightarrow ||\Delta_x D(x)|| \leq 1 \quad for\ all\ x \tag{6.39}$$

有了这个定理，就能够近似地这样去表达 W 距离：

$$W(P_r, P_g) \approx \max_D \{E_{x\sim P_r}[D(x)] - E_{x\sim P_g}[D(x)] - \tag{6.40}$$

$$\lambda E_{x\sim P_{penalty}} max(0, ||\Delta_x D(x))|| - 1)\} \tag{6.41}$$

不需要在整个分布上都满足 Lipschitz 条件，只需要沿着一些直线上的点满足这些，结果就已经很好了，同时在实际中采用的策略也不是取 max，因为不希望 $||\Delta_x D(x)||$ 太小，所以做的是最小化 $(||\Delta_x D(x)|| - 1)^2$，最后改进的 WGAN 就是：

$$W(P_r, P_g) \approx \max_D \{E_{x\sim P_r}[D(x)] - E_{x\sim P_g}[D(x)] - \tag{6.42}$$

$$\lambda E_{x\sim P_{penalty}} (||\Delta_x D(x))|| - 1)^2\} \tag{6.43}$$

改进后的 WGAN 和改进前的 WGAN 相比，训练更加稳定，生成的图片效果也更好。

6.4 应用介绍

GAN 被提出两年以后，很多想法都被研究者们提出、探索并实践，因此催生出了 GAN 的很多应用，下面就来介绍其中的两个应用：Conditonal GAN 和 Cycle GAN。

6.4.1 Conditional GAN

在前面的生成对抗网络学习中，我们发现它能够生成以假乱真的图片，但是这些图片并不一定是我们想要的。比如手写字体的图片生成中，想要生成 4，却生成了 5 或者 9，这使得生成网络并没有实际的作用，因为它生成的图片都是随机的。随着这个问题的出现，研究者开始研究如何能够按照意愿生成图片，这便有了 Conditional GAN。

Conditional GAN 的一个应用是做文字生成图片，比如给出一段文字 “a dog is running”，希望网络生成的图如图 6.13所示。Image caption 就是通过卷积神经网络和循环神经网络给图片加字幕，上面这个过程是它的逆过程。这个逆过程与 image caption 相比难在什么地方呢？

Image caption 通过预训练的卷积神经网络提取图片特征，然后据此特征通过循环神经网络生成文本，比如一张火车的照片，不管火车是什么样的，预训练的卷积神经网络都能够提取有效的特征，据此能够得到文字信息 “train”。但是通过文字生成图片就

面临着一个很大的问题——图片的多样性，比如给出“train”这个单词，需要生成一张火车的图片。

图 6.13 奔跑的狗

图 6.14所示的是 4 张火车的图片。

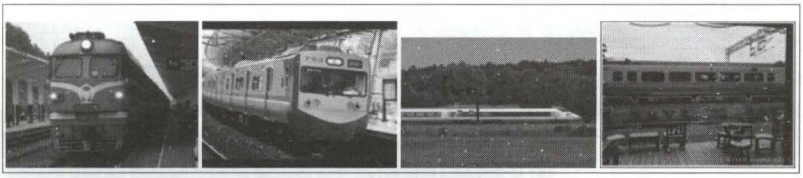


图 6.14 火车

这样并没有很好的办法定义损失函数，如果单纯地将图片作为目标，那么最后的结果就是图 6.14 的平均，全是马赛克，没有具体的内容。

使用生成对抗网络能够比较好地解决问题，这就是 Conditional GAN。原理非常简单，正如它的名字一样，就是普通的生成对抗网络，只是加上一个条件，比如要生成火车的图像，就是将“train”作为一个条件放到普通的生成对抗网络中，可以用图6.15抽象地表示。

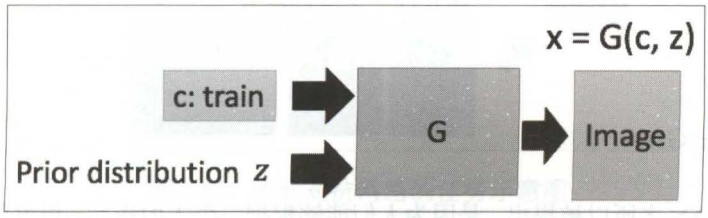


图 6.15 Conditional GAN 的基本思想

以上表示了 Conditional GAN 的基本思想，图6.16引自于 NIPS 2016 的论文 "*Learning What and Where to Draw*"，这张图片展示了具体的生成网络和判别网络。在生成网络中，将给出的文本信息和随机向量拼在一起作为生成网络的输入，接着生成一张图片。在判别网络中，也会将文本信息和中间提取的特征图拼起来，最后输出结果表示真假。其中正样本就是文本和它对应的图片，负样本是文本和生成的图片，以及真实的图片和不对应的文本。

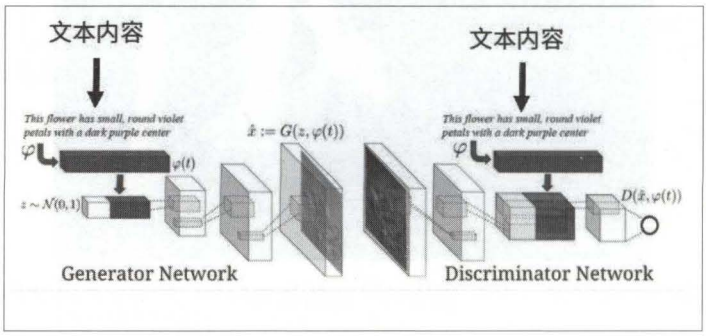


图 6.16　生成网络和判别网络

训练完 Conditional GAN 之后，只需取出生成网络，输入文本作为条件就可以生成想要的图片，效果如图 6.17所示。

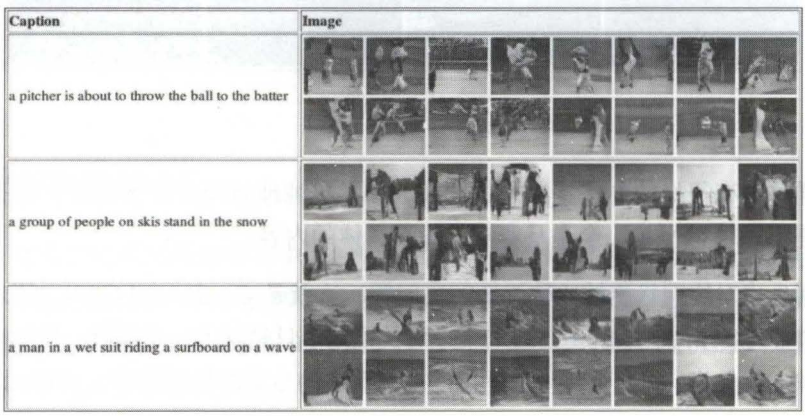


图 6.17　生成的图片

6.4.2　Cycle GAN

Cycle GAN 之所以被提出，是因为人们能够根据一个人的作品，想象他画出的其他场景会是什么样的，也就是说虽然没有对应的数据，但是却能够实现这种对应的映射。

那么机器能否做到这一点呢?

人类之所以能够做到这件事，是因为在两个领域 (X, Y) 之间，存在着一种底层的关系，对于同一个事物，如果有两种不同的映射之后的表达，那么这两种表达之间就是针对同一个事物的一种关联。如果仅使用一种关联对它进行单向映射，那么就无法保证这个映射是单一的，也就是说对于同一个事物，可能会将它映射到两个不同的表达。

针对这个问题，Cycle GAN 提出了一种自然的解决方法，不仅要求单向映射，还要求一种双向的映射，比如 $G: X \to Y, F: Y \to X$ 两个映射，我们希望 $F(G(X)) \approx X, G(F(Y)) \approx Y$。

上面就是 Cycle GAN 提出的原理和动机，先放几张 Cycle GAN 的效果图，如图6.18所示。从图 6.18 可以看出，Cycle GAN 将冬天和夏天互换，将斑马和马进行相互转换。其实这个想法并不是 Cycle GAN 独有的，同时期的 Disco GAN 和 Dual GAN 都有着相似的思想，只是它们的侧重点有所不同。

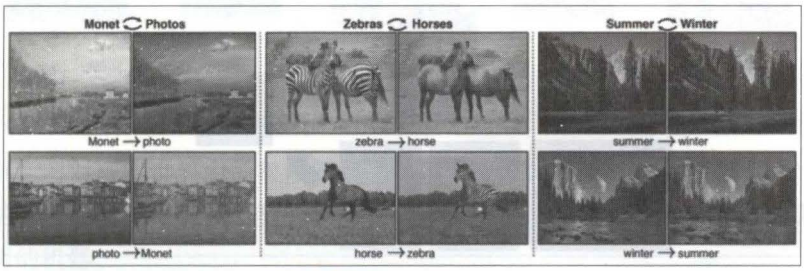


图 6.18 Cycle GAN 效果图

那么具体 Cycle GAN 的网络结构是怎样的呢? 下面先从 GAN 开始，逐渐发展到 Cycle GAN。

根据最基本的生成对抗网络，可以通过图6.19所示的结构来完成所需要的任务。

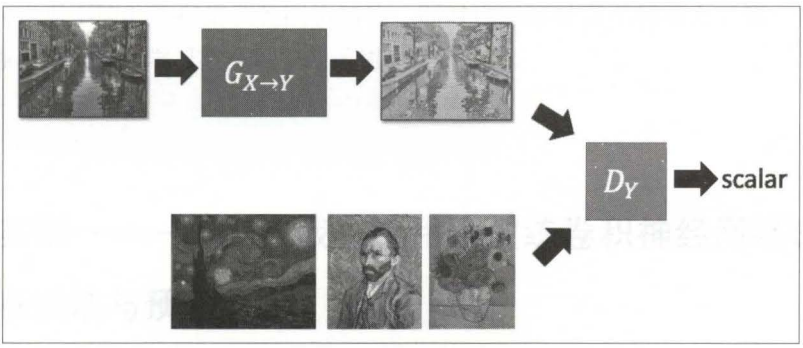


图 6.19 生成对抗网络的任务示意图

将原始图片输入网络，然后通过判别器的判断，使得它的风格尽可能接近我们想

要的风格。但是这样会出现问题，在生成器不断训练的过程中，原始图片通过生成网络会失去信息，变成别的图片。比如图 6.19 输入的是一张河道的图片，可能通过生成网络之后会变成一张梵高的画像，这正是因为单项映射无法保证单一性。

那么 Cycle GAN 是如何做的呢？正如前面介绍的，建立一个双向映射如图6.20所示。

通过建立两个生成网络从而建立一个双向映射，这样就避免了信息的丢失，使得原始图片能够被映射到想要的效果并保留主要信息。

生成对抗网络作为近年来深度学习中升起的新星，近年来被大量的研究者研究，它的变式和应用不仅如此，感兴趣的读者可以进一步阅读相关论文。

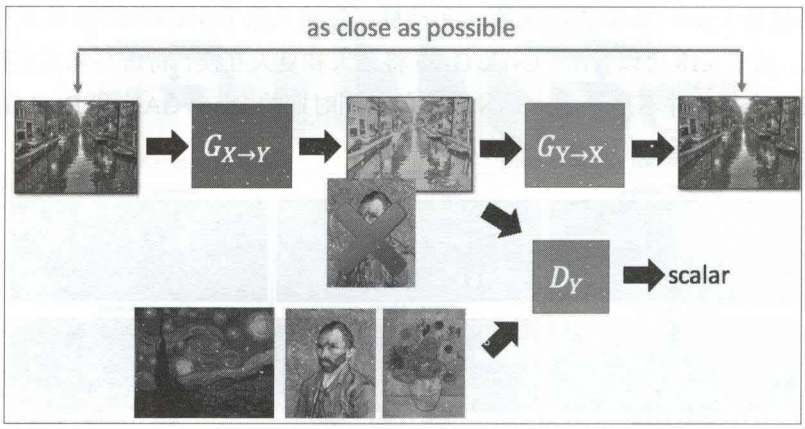


图 6.20　双向映射

本章从生成对抗网络的基本思想开始，接着介绍了数学原理，从数学原理出发推导出它的局限性，从而引入改善的生成对抗网络，最后介绍了生成对抗网络的一些变式和应用，下一章将通过实战讲解具体介绍深度学习在实际问题中的应用。

第 7 章

深度学习实战

机器学习对现实世界中的很多问题都实现了突破性的进展，而机器学习中发展最为迅速的深度学习无疑引领着整个人工智能领域的变革，也许人类因此能够制造出越来越聪明的机器。

前面介绍了深度学习中的基础知识，也实现了一些例子，但是真正要掌握的是：如何在现实中运用深度学习的技术解决实际问题。从数据的收集、模型的搭建、接着不断调参训练模型、最后得到结果，经历完整的过程对深刻地理解深度学习大有裨益。通过实际场景的应用，更加明白深度学习能够处理什么类型的数据，了解其算法的局限，知道如何根据不同的数据集调整网络结构，同时也能够提高工程能力。这一章通过四个实战练习介绍深度学习在实际中的运用。

7.1 实例一——猫狗大战：运用预训练卷积神经网络进行特征提取与预测

这是 Kaggle 在 2013 年的一个比赛，在这个比赛中，需要写一个算法来分辨图7.1中的动物是猫还是狗，这对于人类而言很容易，但是对于机器而言，也许并不简单。

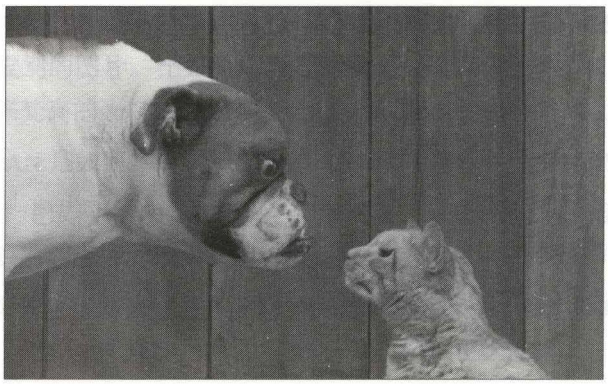


图 7.1　猫与狗

7.1.1　背景介绍

网络保护通常面临一个难题，就是它应该让人能够很容易地识别，而让机器无法识别，比如验证码，它可以有效地减少垃圾邮件，还能够防止用户的密码被恶意破解。

Asirra 是一个图像识别机制的验证码，需要人们根据图片来辨认是猫还是狗，这件事情对于计算机而言是特别难的，但是对于人而言是非常简单且准确的，很多人甚至认为这件事很有趣。Asirra 有很多不同的猫和狗的照片，因为其合作者是一家致力于为流浪动物寻找家园的网站，它们为 Asirra 提供了三百万张猫和狗的图片，接下来要使用它的子集作为训练集。

对于该网站的攻击，随机猜测是其中一种，但是依据图像识别算法能够使得猜测的准确率变得更高。而图片有无穷多种变化，比如背景、角度、光线等，这使得识别变得非常困难。很多年前，计算机视觉的专家给出单张的准确率只有 60%，而且提升变得越来越困难。

到了现在，传统的机器学习算法能够达到 80% 的准确率，但是这并没有达到极限，深度学习中卷积神经网络的出现使得现在的识别任务有更大的提升，下面就以 80% 作为基准，使用卷积神经网络达到更好的效果。

7.1.2　原理分析

对这个问题，根据之前学习的卷积神经网络，可以写一个简单的网络模型，但是会发现自己写的网络效果并没有那么好，不仅准确率不高，甚至有可能收敛速度还很慢，或者不收敛。这个时候，使用一些成熟的模型，比如 VggNet、GoogleNet、ResNet 等就

可以帮助我们解决问题。这些成熟的网络都是由领导全球深度学习技术的实验室做了无数次实验而实现的优秀的网络，而且通过 ImageNet 比赛获得过冠亚军的成绩，所以使用这些网络，对于结果有一定保障。现在深度学习的门槛越来越低，一方面得益于现在的框架让写网络变得很简单；另外一方面就是这些实验室愿意开源共享他们的模型和实验结果。

可以使用现成的模型直接来训练的数据，从已有的模型出发再不断地微调。但是这会带来一些计算资源的问题，因为对大数据集而言，每跑一次实验结果就需要耗费很多计算资源，作为学习者，没有那么强大的资源供我们使用，这个时候是不是就没有办法了呢？其实办法是有的，那就是迁移学习。通过迁移学习，让没有太多计算资源的人也能顺利实现深度学习中复杂模型的训练。

迁移学习

在机器学习的经典监督学习场景中，如果要针对一些任务 A 训练一个模型，会通过提供任务 A 的数据和标签来训练，现在已经在给定的数据集上训练了一个模型 A，并且期望它在同一个任务和未知数据上表现良好。在另外一种情况下，当给定一些任务 B 的数据和标签时，也可以根据任务 B 来训练我们的模型 B，这些都是很合理的，如图 7.2所示。

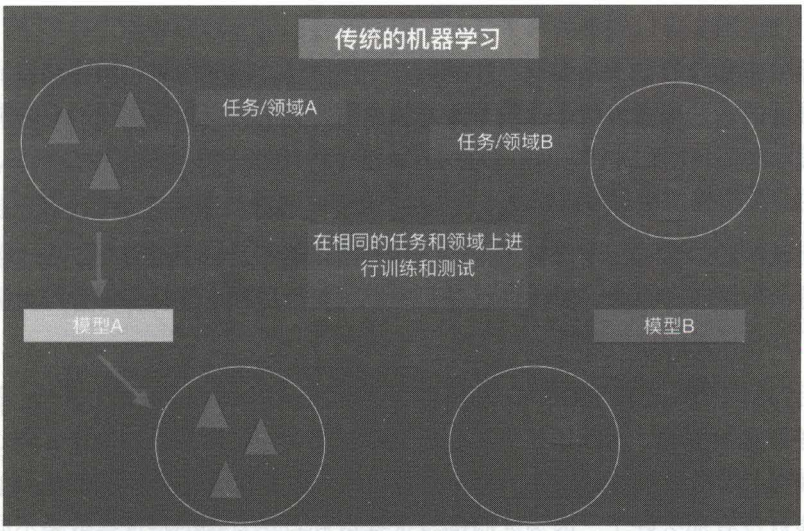


图 7.2 传统的机器学习

但是对于一个特定的任务，如果没有来自于该任务足够的数据集，传统的监督学习就无法支持了。迁移学习允许通过借用已经存在的一些相关任务的标签数据来处理这些场景，把解决相关任务时获得的知识存储下来，并将它应用在我们感兴趣的目标任务中，如图 7.3所示。

正是由于这个原因，我们便可以使用 ImageNet 中预训练好的网络进行迁移学习了，因为网络通过完成 ImageNet 比赛，能够获得知识，也就是网络中的参数，而 ImangeNet 中也有很多猫和狗的图片，所以我们认为网络在 ImageNet 中获得的知识能够进行迁移。

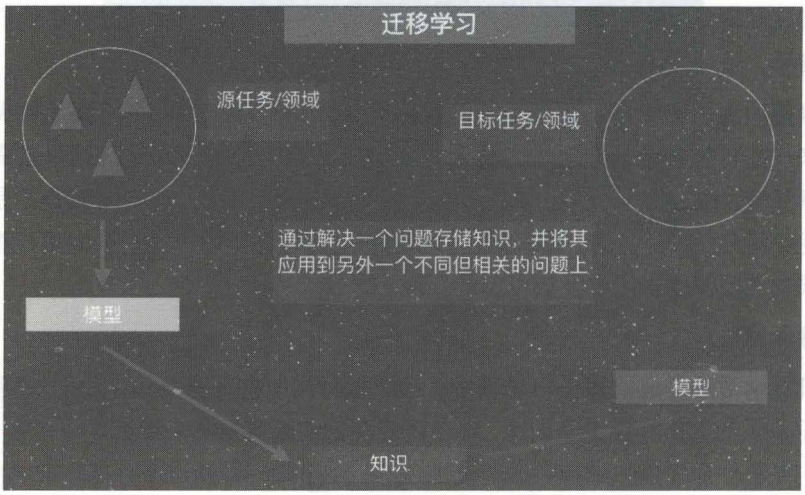


图 7.3　迁移学习

卷积神经网络可以理解为两个部分：前面的卷积部分和后面的分类部分，而前面的卷积部分主要做的事就是提取图片特征，而预训练好的网络对于图片的特征提取效果是非常好的，这是因为网络学到了需要的参数，我们可以直接用预训练的网络卷积部分来提取我们自己图片的特征，而对于我们自己的任务，也就是猫狗二分类，就用我们自己的分类全连接层就可以了。

以上就是迁移学习在图像识别中的一种应用，是不是特别简单呢？总结起来就是将预训练的网络迁移过来，然后训练过程中只更新最后的全连接层部分的参数，实现最后我们自己任务的分类目的。

最后强调一下，迁移学习并不是任何时候都能够使用的，前面我们提过，需要它们完成的任务是相关的，所以迁移学习在相似数据集上的应用效果才是良好的，比如你用的预训练的参数是自然景物的图片分类得到的，那么使用这些参数来做人脸的识别，效果可能就没有那么好了，因为人脸的特征提取和自然景物的特征提取是不同的，所以相应的参数训练后也是不同的。

实现方法

在代码实现中，我们会层层递进。第一种方法是导入预训练的卷积网络，将最后的全连接层改成我们自己设计的全连接层，然后更新整个网络的参数，最后能特别快地达到收敛；第二种方法是锁定前面卷积层的参数，让网络训练只更新最后全连接的参

数，这样可以让我们训练时间大大减少；前面两种方法都是用一种预训练好的网络，第三种方法中我们使用多个预训练好的网络，将它们并联在一起，图片经过每个网络都会得到特征图，我们将这些特征图拼接在一起进入最后的全连接层。

7.1.3 代码实现

1. 数据预处理

数据集可以去 https://www.kaggle.com/c/dogs-vs-cats/data 下载，并解压整个文件，因为里面的猫和狗的图片是混在一起的，所以我们需要通过预处理将它们分成两个文件夹。

我们先创建两个文件夹：一个是训练集文件夹，一个是测试集文件夹。这两个文件夹内部都有两个文件夹：一个文件夹中放狗的图片，一个文件夹中放猫的图片。

接着我们将猫和狗的图片分别移动到相应的文件夹中：

```
data_file = os.listdir('/media/sherlock/Files/kaggle_dog_vs_cat/zip')
dog_file = list(filter(lambda x: x[:3]=='dog', data_file))
cat_file = list(filter(lambda x: x[:3]=='cat', data_file))
```

我们将解压之后所有的图片放到 zip 文件夹中，第一行代码是提取出文件夹中的所有图片的名称，然后第二行和第三行代码分别是提取出狗和猫的图片，其中使用了 `filter`，这个函数能够将满足条件的数据提取出来，上面就是将图片名字是 dog 和 cat 的图片分别取出来，存为两个 list。

接着我们将猫和狗的图片分别移动到训练集和验证集中，其中 90 % 的数据作为训练集，10 % 的图片作为验证集，使用 `shutil.move()` 来移动图片。

```
root = '/media/sherlock/Files/kaggle_dog_vs_cat/'
for i in range(len(dog_file)):
    pic_path = root + 'zip/' + dog_file[i]
    if i < len(dog_file)*0.9:
        obj_path = train_root + '/dog/' + dog_file[i]
    else:
        obj_path = val_root + '/dog/' + dog_file[i]
    shutil.move(pic_path, obj_path)

for i in range(len(cat_file)):
    pic_path = root + 'zip/' + cat_file[i]
```

```
    if i < len(dog_file)*0.9:
        obj_path = train_root + '/cat/' + cat_file[i]
    else:
        obj_path = val_root + '/cat/' + cat_file[i]
    shutil.move(pic_path, obj_path)
```

这样就完成数据预处理部分了，可以用可视化图片来帮助我们理解训练数据，如图7.4所示。

接下来我们就可以开始迁移学习的模型训练了。

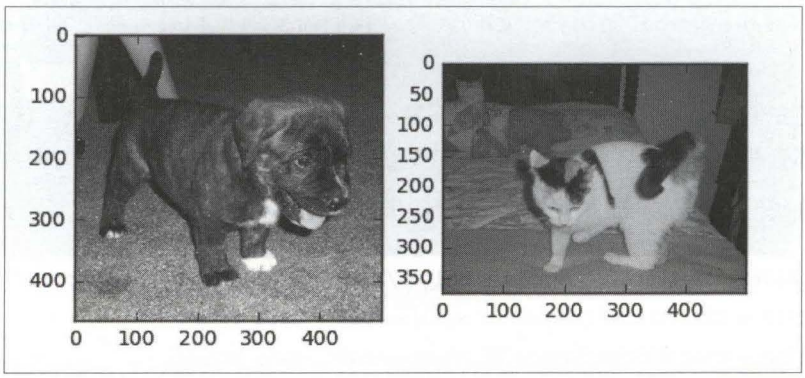


图 7.4　迁移学习的模型训练

2. 方法一

首先我们使用第一种方法进行迁移学习，对于预训练权重的模型，PyTorch 将常见的模型和预训练的参数都已经存在了网上，所以我们不需要自己再去写网络在 ImageNet 上训练模型了，所有预训练的模型都在 `torchvision.models` 里面，可以通过下面这样定义预训练的模型。

```
transfer_model = models.resnet18(pretrained=True)
dim_in = transfer_model.fc.in_features
transfer_model.fc = nn.Linear(dim_in, img_classes)
```

这里使用的网络是一个简单的 18 层的残差网络（ResNet），将最后的全连接层换成了适用于我们数据集分类的全连接层。

接着就可以开始训练了，可以发现经过一次数据迭代，训练集准确率就达到了 88% 左右，验证集准确率都达到了 97% 左右，使用没有预训练的网络第一次迭代之后准确率应该只有 60% 左右，从中可以看出使用迁移学习的好处，能够极大地减少计算资源。最后经过 10 次迭代，验证集能够达到 98.7% 的准确率，如果希望实现更高的准确率，

则可以使用更复杂的预训练模型，然后调整全连接层的参数，并且使用更加复杂的图像增强，结果如图7.5所示。

```
1/10
**********
Train
Loss: 0.499258, Acc: 0.7588
Loss: 0.410139, Acc: 0.8155
Loss: 0.361413, Acc: 0.8425
Loss: 0.329575, Acc: 0.8580
Loss: 0.308781, Acc: 0.8676
Loss: 0.290153, Acc: 0.8774
Loss: 0.276034, Acc: 0.8833
Loss: 0.275705, Acc: 0.8836, Time: 83s
Validation
Loss: 0.067554 Acc: 0.9776
```

```
10/10
**********
Train
Loss: 0.125356, Acc: 0.9466
Loss: 0.122586, Acc: 0.9483
Loss: 0.122460, Acc: 0.9494
Loss: 0.119496, Acc: 0.9494
Loss: 0.117418, Acc: 0.9500
Loss: 0.115596, Acc: 0.9513
Loss: 0.114691, Acc: 0.9519
Loss: 0.114711, Acc: 0.9519, Time: 85s
Validation
Loss: 0.032444 Acc: 0.9876
```

图 7.5 方法一的训练结果

3. 方法二

方法二跟方法一本质上是相同的，只是方法二会固定住卷积层的参数，只更新全连接层的参数，这在 PyTorch 中的实现也很简单。

```
for param in transfer_model.parameters():
        param.requires_grad = False

optimizer = optim.SGD(transfer_model.fc.parameters(), lr=1e-3)
```

首先提取出模型中的参数，然后将其设定为不求梯度，最后在优化器中设置需要优化的参数只有全连接层的参数，要注意不能将卷积层的参数也放进优化器中，不然会报错，这样就只需更新最后的全连接层参数。

因为不需要更新卷积层的参数，所以训练的时间可以大为缩短，同时验证集的准确率一开始就能够达到 96%，但是有得必有失，网络更新的参数变少，这导致经过 10 次迭代验证集的精度只有 97%，没有所有参数都进行更新的精度高，结果如图 7.6所示。

```
1/10
**********
Train
Loss: 0.621323, Acc: 0.6606
Loss: 0.511330, Acc: 0.7550
Loss: 0.457697, Acc: 0.7908
Loss: 0.419723, Acc: 0.8137
Loss: 0.392866, Acc: 0.8284
Loss: 0.372802, Acc: 0.8389
Loss: 0.357976, Acc: 0.8468
Loss: 0.357414, Acc: 0.8470, Time: 36s
Validation
Loss: 0.136617 Acc: 0.9628
```

```
10/10
**********
Train
Loss: 0.175730, Acc: 0.9191
Loss: 0.186050, Acc: 0.9166
Loss: 0.188697, Acc: 0.9163
Loss: 0.186911, Acc: 0.9178
Loss: 0.185488, Acc: 0.9195
Loss: 0.184409, Acc: 0.9203
Loss: 0.185115, Acc: 0.9190
Loss: 0.185138, Acc: 0.9190, Time: 35s
Validation
Loss: 0.067383 Acc: 0.9736
```

图 7.6 方法二的训练结果

4. 方法三

在第三种方法中，我们组合了多个预训练网络，都将它们的卷积层参数固定，只更新最后的全连接层的参数，相当于是方法二的衍生，在每一次迭代中，都需要将图片前

向传播，通过卷积层，到全连接层，最后输出结果，接着进行反向传播更新全连接层的参数。如果数据集特别大，这样做效率特别低，因为卷积层的参数没有更新，所以每次迭代中数据集前向传播经过卷积层的结果是一样的，所以没有必要每次都进行前向传播，只需要将数据集一次迭代中前向传播经过卷积网络的结果保存起来就可以了，这个结果成为特征向量。

首先定义好特征提取的预训练网络如下：

```
class feature_net(nn.Module):
    def __init__(self, model):
        super(feature_net, self).__init__()

        if model == 'vgg':
            vgg = models.vgg19(pretrained=True)
            self.feature = nn.Sequential(*list(vgg.children())[:-1])
            self.feature.add_module('global average', nn.AvgPool2d(9))
        elif model == 'inceptionv3':
            inception = models.inception_v3(pretrained=True)
            self.feature = nn.Sequential(*list(inception.children())[:-1])
            self.feature._modules.pop('13')
            self.feature.add_module('global average', nn.AvgPool2d(35))
        elif model == 'resnet152':
            resnet = models.resnet152(pretrained=True)
            self.feature = nn.Sequential(*list(resnet.children())[:-1])

    def forward(self, x):
        """
        model includes vgg19, inceptionv3, resnet152
        """
        x = self.feature(x)
        x = x.view(x.size(0), -1)
        return x


class classifier(nn.Module):
    def __init__(self, dim, n_classes):
        super(classifier, self).__init__()
        self.fc = nn.Sequential(
```

```
            nn.Linear(dim, 1000),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(1000, n_classes)
        )

    def forward(self, x):
        x = self.fc(x)
        return x
```

feature_net 就是特征提取的网络，接受的参数 model 有 vgg，inceptionv3，resnet152，分别表示用的是 19 层 Vgg 网络，Inceptionv3 网络或者 152 层的残差网络作为预训练的网络，然后将网络的全连接层去掉，最后在特征提取的卷积层的最后一层加上一个平均池化层，将结果转化为特征向量。

classifier 是对于我们自己的数据集的全连接分类层，最后的分类结果是两类，其中使用 Dropout 防止过拟合。

然后就可以将数据输入到网络中进行前向传播得到特征向量，对于训练集和验证集都需要进行前向传播，一共要使用三个预训练的模型，最后可以将结果存成 h5 文件。

```
featurenet = feature_net(model)
    if use_gpu:
        featurenet.cuda()
    feature_map = torch.FloatTensor()
    label_map = torch.LongTensor()
    for data in tqdm(dataloader[phase]):
        img, label = data
        if use_gpu:
            img = Variable(img, volatile=True).cuda()
        else:
            img = Variable(img, volatile=True)
        out = featurenet(img)
        feature_map = torch.cat((feature_map, out.cpu().data), 0)
        label_map = torch.cat((label_map, label), 0)
    feature_map = feature_map.numpy()
    label_map = label_map.numpy()
    file_name = '_feature_{}.hd5f'.format(model)
    h5_path = os.path.join(outputPath, phase) + file_name
```

```
    with h5py.File(h5_path, 'w') as h:
        h.create_dataset('data', data=feature_map)
        h.create_dataset('label', data=label_map)
```

feature_net 是前面定义的特征提取网络，将每种网络输出的特征向量保存到 h5 文件中，注意在提取特征向量的时候，数据是不能随机打乱的，因为使用多个模型，每次都随机打乱就会造成标签混乱，然后将 h5 文件保存到当前目录下，最后通过下面的方式将特征提取出来，定义一个由特征向量组成的数据集。

```
class h5Dataset(Dataset):

    def __init__(self, h5py_list):
        label_file = h5py.File(h5py_list[0], 'r')
        self.label = torch.from_numpy(label_file['label'].value)
        self.nSamples = self.label.size(0)
        temp_dataset = torch.FloatTensor()
        for file in h5py_list:
            h5_file = h5py.File(file, 'r')
            dataset = torch.from_numpy(h5_file['data'].value)
            temp_dataset = torch.cat((temp_dataset, dataset), 1)

        self.dataset = temp_dataset

    def __len__(self):
        return self.nSamples

    def __getitem__(self, index):
        assert index < len(self), 'index range error'
        data = self.dataset[index]
        label = self.label[index]
        return (data, label)
```

这个数据集的定义在前面 PyTorch 基础部分已经讲过了，如果忘了，可以去前面的部分看一下，它是继承于 torch.utils.data.Dataset 这个基类，只需要简单地将每个 h5 文件打开，然后提取出每张图片使用不同的预训练网络得到的相应的特征向量，将它们拼接成一个特征向量就可以了。每种网络大概需要 10 到 20 分钟导出特征向量，如果不能使用 GPU 加速，那么时间还会更长，电脑配置不高的读者可以下载我已经导出的特征向量，在百度网盘中，地址是 https://pan.baidu.com/s/1c2lci1U。

最后我们使用这些特征向量来训练一个 2 层的全连接网络就可以了，可以发现收敛特别快，而且得到的结果特别好，一次迭代之后在训练集上的准确率是 95%，在验证集上能够达到 99.6% 的准确率，因为验证集上的数据较少，所以得到的准确率也相当高，最后在训练完 20 次迭代之后，训练集的准确率是 99.52%，验证集的准确率是 99.68%，这大大好于之前只是用一个小型预训练网络的结果，同时训练所需要的时间也大大减少，每次迭代只需要 1 秒钟就能够完成，这个可以看作将三个训练好的网络集成在一起构成一个大的网络，结果如图7.7所示。

```
1
**********
Train
Loss: 0.489418, Acc: 0.820625
Loss: 0.299524, Acc: 0.892344
Loss: 0.217856, Acc: 0.923646
Loss: 0.177051, Acc: 0.937891
Loss: 0.151155, Acc: 0.946625
Loss: 0.133573, Acc: 0.952708
Loss: 0.118804, Acc: 0.957991
Loss: 0.118397, Acc: 0.958089, Time: 1s
Validation
Loss: 0.018084, Acc: 0.996000
```

```
20
**********
Train
Loss: 0.011539, Acc: 0.995625
Loss: 0.010487, Acc: 0.995938
Loss: 0.011226, Acc: 0.996354
Loss: 0.010757, Acc: 0.996250
Loss: 0.011719, Acc: 0.995687
Loss: 0.012612, Acc: 0.994948
Loss: 0.012252, Acc: 0.995179
Loss: 0.012231, Acc: 0.995200, Time: 1s
Validation
Loss: 0.008222, Acc: 0.996800
```

图 7.7 方法三的训练结果

7.1.4 总结

通过这个实例的学习，我们了解到如何从数据开始进行分析，然后到模型的建立，最后进行结果分析的整个建模流程。我们掌握了迁移学习的基本思想，同时学会了如果使用预训练的网络进行微调来提高网络准确率，以及如何使用卷积网络的对数据进行特征提取，也真切地感受到了迁移学习对计算资源的节省。

7.2 实例二——Deep Dream：探索卷积神经网络眼中的世界

在前面一个实例中我们使用了预训练的卷积神经网络实现了迁移学习，而预训练的卷积网络应用不仅仅于此，2015 年 Google 发布了一个很有意思的东西，叫做 Deep Dream，网上瞬间掀起了 Deep Dream 的热潮，各种各样有着 Deep Dream 效果的图片漫天飞，就算你没有听过 Deep Dream，你也一定已经看过下面的图片了，如图7.8所示。

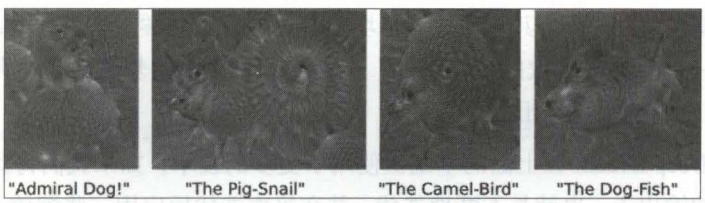


图 7.8 云朵梦境

这些图片都是天空中云朵的图片，但是却呈现出幻觉和梦境，所以这个算法被称为 Deep Dream。这个算法其实是意外得到的结果。

7.2.1 原理介绍

我们知道神经网络在图像分类上取得了显著的进展，但是由于深度学习网络中参数太多了，导致这个算法是一个黑盒子，虽然能够达到良好的效果，但是人们仍然对其内部知之甚少，所以人们希望能够窥探一下网络里面的内部。

1. 反向神经网络

一个神经网络读入一张图片，通过多层网络，最后输出一个分类的结果，但是我们仅仅知道一个结果并不够，神经网络的一个挑战是要理解在每一层到底都发生了什么事。我们知道经过训练之后，每一层网络逐步提取越来越高级的图像特征，直到最后一层将这些特征比较做出分类的结果。比如前面几层也许在寻找边缘和拐角的特征，中间几层分析整体的轮廓特征，这样不断的增加层数就可以发展出越来越多的复杂特征，最后几层将这些特征要素组合起来形成完整的解释，这样到最后网络就会对非常复杂的东西，比如树叶、小猫等图片有了反应。

为了理解神经网络是如何学习的，我们必须要理解特征是如何被提取和识别的。如果分析一些特定层的输出，就可以发现当它识别到了一些特定的模式，它就会将这些特征显著地增强，而且层数越高，识别的模式就越复杂。当我们分析这些神经元的时候，输入很多图片，然后去理解这些神经元到底检测出了什么特征是不现实的，因为很多特征人眼是很难识别的。一个更好的办法是将神经网络颠倒一下，不是输入一些图片去测试神经元提取的特征，而是我们选出一些神经元，看它能够模拟出最可能的图片是什么，将这些信息反向传回网络，每个神经元将会显示出它想增强的模式或者特征。

比如从图 7.8 中我们能够看出不同的神经元模拟出了不同的增强特征和模式，有一些是狗，有一些是蜗牛，还有一些是鱼。

2.Deep Dream

通过上面的过程我们会迫使神经网络在图片中产生一些本来不存在的东西，这也就产生了类似梦境和幻觉，其实上这些梦境强调了网络到底学习到了什么，这种技术给我们提供了一种对抽象层次的定性感受，虽然这和现实中的梦境没有太大的关系，这也就是 Deep Dream 最早提出的灵感。

实际上 Deep Dream 在上面的基础上使用了更多的技术，如果我们将此算法反复地应用在自身的输出上，也就是不断地迭代，并在每次迭代后应用一些缩放，这样我们就能够不断地激活特征，得到无尽的新效果，比如最开始网络的一些神经元模拟出了一

张有狗的轮廓的图，通过不断的迭代，网络就会越来越相信这是一只狗，图片中狗的样子也就会越来越明显。

7.2.2 预备知识：backward

前面讲了 Deep Dream 的基本原理，在实现之前，我们需要掌握一个预备知识，就是 PyTorch 中的 backward。

前面我们已经写了很多网络，也对网络进行了多次训练，这个过程都特别简单、直观，但是一个网络训练过程中的操作一直没有仔细去考虑过，那就是 `loss.backward()`，看到这个大家一定都很熟悉，loss 是网络的损失函数，是一个标量，但是如果 loss 不是一个标量，而是一个向量，那么 `loss.backward()` 是什么结果呢？

我们可以去试试，写一个简单的程序：

```
import torch
from torch.autograd import Variable
x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 1
y.backward()
```

运行一下程序，会显示错误如图 7.9所示。

```
    'backward should be called only on a scalar (i.e. 1-element tensor) '
RuntimeError: backward should be called only on a scalar (i.e. 1-element tensor) o
r with gradient w.r.t. the variable
```

图 7.9 运行错误

我们来读一读这个错误是什么意思。backward 只能被应用在一个标量上，也就是一个 1×1 的张量上，或者在 backward 中传入跟变量相关的梯度。

之前一直都是将 backward 应用在一个标量，对于向量我们还没有碰到过。为什么这里我们需要知道 backward 是如何作用在向量上的呢？这是因为在 Deep Dream 中需要对向量进行反向传播。

首先复习一下标量的反向传播。

```
# simple gradient
a = Variabel(torch.FloatTensor([2, 3]), requires_grad=True)
b = a + 3
c = b * b * 3
out = c.mean()
```

```
out.backward()
print('*'*10)
print('=====simple gradient======')
print('input')
print(a.data)
print('compute result is')
print(out.data[0])
print('input gradients are')
print(a.grad.data)
```

上面这个过程运算很简单，可以把数学表达式写出来，传入的参数 $x_1 = 2, x_2 = 3$，特别注意 Variable 里面默认的参数 `requires_grad=False`，所以这里要重新传入 `requires_grad=True`，让它成为计算图中的一个叶子节点。

$$a = (x_1, x_2) \tag{7.1}$$

$$b = (x_1 + 3, x_2 + 3) \tag{7.2}$$

$$c = (3 \times (x_1 + 3)^2, 3 \times (x_2 + 3)^2) \tag{7.3}$$

$$out = \frac{3 \times ((x_1 + 3)^2 + (x_2 + 3)^2)}{2} \tag{7.4}$$

那么我们对其求偏导也很简单：

$$\frac{\partial out}{\partial x_1} = 3(x_1 + 3)|_{x_1=2} = 15 \tag{7.5}$$

$$\frac{\partial out}{\partial x_2} = 3(x_2 + 3)|_{x_2=3} = 18 \tag{7.6}$$

我们就这样依靠简单的微积分知识算出结果，运行一下程序，得到如图7.10所示的结果，和程序结果是一致的。

下面我们来学习一下如何能够对非标量的情况下使用 backward。

```
m = v(t.FloatTensor([[2, 3]]), requires_grad=True)
n = v(t.zeros(1, 2))
n[0, 0] = m[0, 0] ** 2
n[0, 1] = m[0, 1] ** 3
```

```
**********
=====simple gradient======
input

 2
 3
[torch.FloatTensor of size 2]

compute result is
91.5
input gradients are

 15
 18
[torch.FloatTensor of size 2]
```

图 7.10 标量的反向传播结果

首先定义好输入 $m=(x_1,x_2)=(2,3)$，然后做的操作就是 $n=(x_1^2,x_2^3)$，这样就定义好了一个向量输出，得到的结果第一项只和 x_1 有关，第二项只和 x_2 有关，那么求解这个梯度，我们知道 $\frac{\partial n_1}{\partial x_1}=2x_1=4,\frac{\partial n_2}{\partial x_2}=3x_2^2=27$，下面开始探究如何能够让它调用 backward。

首先想到的是里面的参数是要求梯度的对象，我们这样调用 `n.backward(m.data)`，会得到如图7.11所示的结果。

```
input gradients are

  8  81
[torch.FloatTensor of size 1x2]
```

图 7.11 非标量调用 backward 结果

得到的结果跟之前计算出来的结果不一样，我们计算得到的结果是 4 和 27，PyTorch 反向传播得到的结果是 8 和 81，传入的参数是 `m.data`，数值是 (2, 3) 的向量，我们希望得到的梯度结果是 (4, 27)，注意到 $4\times 2=8,27\times 3=81$，其实 backward 将传入的参数 `m.data` 每个元素分别乘上了得到的梯度，所以我们传入 `n.backward(t.FloatTensor([[1, 1]]))`，可以得到如图 7.12所示的结果。

```
input gradients are

  4  27
[torch.FloatTensor of size 1x2]
```

图 7.12 结果纠正

这就跟我们计算的结果一样了。

下面来试试另外一种情况：

```
m = Variable(torch.FloatTensor([[2, 3]]), requires_grad=True)
j = torch.zeros(2 ,2)
k = Variable(torch.zeros(1, 2))
m.grad.data.zero_()
k[0, 0] = m[0, 0] ** 2 + 3 * m[0 ,1]
k[0, 1] = m[0, 1] ** 2 + 2 * m[0, 0]
```

将上面的代码写成数学表达式就是 $m=(x_1=2,x_2=3), k=(x_1^2+3x_2, x_2^2+2x_1)$，直接对 k 反向传播，传入参数 `k.backward(t.FloatTensor([[1, 1]]))`，可以得到如图 7.13所示的结果。

```
m.grad.data

 6  9
[torch.FloatTensor of size 1x2]
```

图 7.13　直接对 k 反向传播

下面我们手动算一算结果是什么。$\frac{\partial(x_1^2+3x_2)}{\partial x_1}=2x_1=4$, $\frac{\partial(x_1^2+3x_2)}{\partial x_2}=3$, $\frac{\partial(x_2^2+2x_1)}{\partial x_1}=2$, $\frac{\partial(x_2^2+2x_1)}{\partial x_2}=2x_2=6$，手动算出来是上面四个结果，这和上面输出的结果是不一样的，上面只输出了两个结果，而且数值还不对。其实这是因为我们没有正确理解传入参数是如何作用的，下面来解释一下 PyTorch 是如何得到上面这个结果的。

已经知道得到的 $k=(k_1,k_2)$，以及传入的参数是 1 和 1，其中第一个结果是通过 $1\times\frac{dk_1}{dx_1}+1\times\frac{dk_2}{dx_1}=2x_1+2=6$ 得到的，第二个结果是通过 $1\times\frac{dk_1}{dx_2}+1\times\frac{dk_2}{dx_2}=3+2x_2=9$ 得到的，这样就理解了 PyTorch 中 backward 是如何处理传入的参数。

下面通过 PyTorch 来求出上面计算所得到的 4 个结果，这 4 个结果又称为 jacobian 矩阵。

```
# jacobian
j = torch.zeros(2 ,2)
k = Variable(torch.zeros(1, 2))
m.grad.data.zero_()
k[0, 0] = m[0, 0] ** 2 + 3 * m[0 ,1]
k[0, 1] = m[0, 1] ** 2 + 2 * m[0, 0]
k.backward(t.FloatTensor([[1, 0]]), retain_variables=True)
```

```
j[:, 0] = m.grad.data
m.grad.data.zero_()
k.backward(t.FloatTensor([[0, 1]]))
j[:, 1] = m.grad.data
print('jacobian matrix is')
print(j)
```

通过运行上面的程序，可以得到如图7.14所示的结果。

```
jacobian matrix is

 4  2
 3  6
[torch.FloatTensor of size 2x2]
```

图 7.14 jacobian 矩阵

这里我们使用了 backward() 里面另外的一个参数 retain_variables=True，这个参数默认是 False，表示的意思是反向传播之后这个计算图的内存会被释放，如果计算图被释放掉，就没办法进行第二次反向传播了，所以我们需要将其设置为 True，因为这里我们需要进行两次反向传播求得 jacobian 矩阵。

最后再举一个矩阵乘法的例子检验一下计算结果，下面是代码，留给读者自己去学习以掌握本小节的内容。

```
x = torch.FloatTensor([2, 1]).view(1, 2)
x = Variable(x, requires_grad=True)
y = Variabel(torch.FloatTensor([[1, 2], [3, 4]]))

z = torch.mm(x, y)
jacobian = torch.zeros((2, 2))
z.backward(torch.FloatTensor([[1, 0]]), retain_variables=True)  # dz1/dx1, dz2/dx1
jacobian[:, 0] = x.grad.data
x.grad.data.zero_()
z.backward(torch.FloatTensor([[0, 1]]))  # dz1/dx2, dz2/dx2
jacobian[:, 1] = x.grad.data
print('=========jacobian========')
print('x')
print(x.data)
```

```
print('y')
print(y.data)
print('compute result')
print(z.data)
print('jacobian matrix is')
print(jacobian)
```

7.2.3 代码实现

前面两个小节已经讲了 Deep Dream 的基本原理和预备知识 backward 了，下面为了更清晰地了解 Deep Dream，我们用代码来实现一下。

1. 实现原理

预训练的网络对特定的分类任务有良好的效果，为了探究它到底学到了什么，直接理解神经元提取的特征是很困难的，这时可以将一些与任务无关的图片输入，希望通过网络对其提取特征，然后反向传播的时候不再更新网络的参数，而是更新图片中的像素点，不断地迭代让网络越来越相信这张图片属于分类任务中的某一类。这就是 Deep Dream 实现的原理，看上去是非常简单的，但是实际运用中要应用一些训练技巧才能达到更好的效果。

2. 预训练网络

首先需要一个预训练的网络，同时还需要对预训练网络中的 `foward` 函数进行调整，因为每次我们并不是对整个网络进行前向传播，而是得到网络的中间输出结果，同时希望能够很自由地控制输出层。在 PyTorch 中这个实现是很简单的，我们使用预训练的 50 层的残差网络（ResNet），PyTorch 已经有 50 层残差网络的实现，只需要对其做一些修改就能满足要求。

```
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
```

```
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1, bias=False
        )
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out

class CustomResNet(models.resnet.ResNet):
    def forward(self, x, end_layer):
        """
        end_layer range from 1 to 4
        """
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
```

```
        x = self.maxpool(x)

        layers = [self.layer1, self.layer2, self.layer3, self.layer4]
        for i in range(end_layer):
            x = layers[i](x)
        return x


def resnet50(pretrained=False, **kwargs):
    model = CustomResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet50']))
    return model
```

首先定义一个最基本的残差模块，然后定义个性化的 ResNet 模块，这个模块是继承于 PyTorch 中的 ResNet 模块，不需要改变其初始化，让它去继承 ResNet 的初始化，只需要重新定义 `forward` 就可以，其中我们加入了结束层，也就是我们希望网络输出第几层的结果。

3. 训练 Deep Dream

定义好了预训练的网络，接下来就可以开始训练模型，再次说明，网络中的参数是预训练的参数，不发生改变，更新的是图片中的像素点。

对于分类问题，使用交叉熵作为损失函数，那么在 Deep Dream 中使用什么作为损失函数呢？非常简单，之前讲过 Deep Dream 希望能够在迭代中不断地让网络更加确定这个图片属于某一类，所以损失函数就是网络结束层输出的特征向量的 L2 范数，目标是最大化 L2 范数来使得图片经过网络之后提取的特征更像网络希望提取的特征。

看着可能有点难以理解，举个例子，一朵云提取特征之后有一点点像狗的图片提取的特征，因为网络的参数不会改变，所以图片如果不改变，那么再次输入网络之后得到的特征还是跟之前一样，只是有一点点像狗的图片提取出来的特征。Deep Dream 希望通过反向传播更新图片的像素点，使得提取出来的特征更大，也就是使得提取出来的特征跟狗的图片提取出来的特征更像，那么用 L2 范数作为损失函数，网络不断更新图片的像素点来最大化 L2 范数，最终使得提取的特征越来越大，我们将多次更新之后的图片输出，也就得到了 Deep Dream 效果之后的图片。

为了得到更好的图片，在训练中需要应用一些小的技巧，否则得到的图片可能会存在很多噪声，或者需要很长的时间才能达到满意的效果。

首先，对于输入的图片，我们需要对其做一些随机抖动，下面是示例代码。

```
shift_x, shift_y = np.random.randint(-max_jitter, max_jitter + 1, 2)
img = np.roll(np.roll(img, shift_x, -1), shift_y, -2)
```

max_jitter 是一个整数，表示抖动的范围，随机从中取出两个整数表示 x 轴和 y 轴的抖动程度，然后使用 np.roll 对数组沿着一个维度进行平移，上面的代码首先对图片的第三个维度进行平移 shift_x，然后对第二个维度进行平移 shift_y。

如何进行反向传播呢，是不是需要计算出 L2 范数呢？其实不需要，L2 范数的公式如下：

$$||x||_{L^2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \tag{7.7}$$

对这个结果进行求导是很麻烦的，因为里面有根号，非常影响计算效率，而且目标是最大化 L2 范数，所以去掉根号不影响计算的结果。去掉根号之后求导就变得容易了，可以得到下面的式子：

$$||x||_{L^2}^2 = x_1^2 + x_2^2 + \cdots + x_n^2 \tag{7.8}$$

每个参数 x 都是图片像素点的函数，记图片中像素点为 $p_1, p_2, \cdots p_k$，那么 $x_i = x_i(p_1, p_2, \cdots p_k)$，所以对其求偏导，可以得到下面的结果：

$$\frac{\partial ||x||_{L^2}^2}{\partial p_j} = 2\sum_{i=1}^{n} x_i \frac{\partial x_i}{\partial p_j} \tag{7.9}$$

前面的系数 2 是常数，可以去掉，这样就得到了反向传播时候的梯度的计算方法。这里就要用到上一节讲的 PyTorch 中 backward，可以通过下面的方式得到所有待更新的像素点 p_j 的梯度。

```
act_value = model.forward(img_variable, end_layer)
act_value.backward(act_value.data)
```

act_value.backward(act_value.data) 就表示 $\sum_{i=1}^{n} x_i \frac{\partial x_i}{\partial p_j}$，这就是上面 L2 范数反向传播的公式。

第二个训练技巧是对学习率进行一些限制，首先将所有参数梯度的绝对值求平均值，然后用学习率除以均值，这样就得到了实际用的学习率。

```
ratio = np.abs(img_variable.grad.data.cpu().numpy()).mean()
learning_rate_use = learning_rate / ratio
```

```
3  img_variable.data.add_(img_variable.grad.data * learning_rate_use)
```

最后还需要使用一个小技巧，就是使用多尺度的图片进行计算，如果一直使用原始的图片，可能收敛速度会比较慢，所以我们先将图片缩小进行更新，再放大进行更新。

```
1  for i in range(octave_n - 1):
2      octaves.append(nd.zoom(octaves[-1], (1, 1, 1.0 / octave_scale, 1.0 /
       octave_scale), order=1))
```

octave_n 表示有多少张小图片，使用 scipy.ndimage.zoom 进行图片的缩放，每次缩放的比例是 1/octave_scale，这样就可以得到大小递减的小图片，然后从小到大依次对图片的像素点进行更新，最后得到 Deep Dream 的图片。

最后做完一次更新需要将图片做逆抖动，开始的时候图片经过平移，现在将图片反向平移回来，同时需要对图片的像素点进行裁剪以控制大小。

```
1  img = np.roll(np.roll(img, -shift_x, -1), -shift_y, -2)
2  img[0, :, :, :] = np.clip(img[0, :, :, :], -mean / std, (1 - mean) / std)
```

4.Deep Dream 结果

对一张云的图片输入做 Deep Dream，经过 Deep Dream 的效果之后变成了图7.15，可以看到图片中多了一些狗头，还有一些眼睛，中间左边有一个明显的塔。

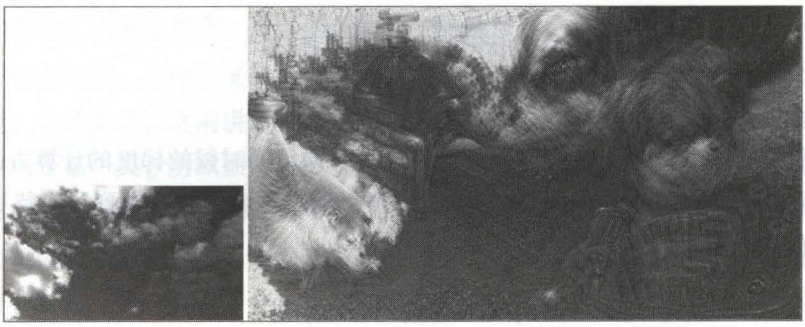


图 7.15　对云做 Deep Dream

除此之外，还可以控制 Deep Dream 中的梦境，也就是说可以控制图片中出现的东西。要实现其实很简单，之前最大化特征向量的 L2 范数，这导致了图片中出现了各种各样的图片，所以要实现梦境控制，我们需要修改一下目标函数。

首先需要输入一张图片作为梦境的控制图片，将控制图片通过网络前向传播得到其特征向量，然后将原始图片输入网络也得到原始图片的特征向量，这两个特征向量的大小不同，所以先将它们重新排列成新的矩阵，然后做矩阵乘法，最后选择矩阵乘法

里面最大的下标，这些下标对应于原始图片和控制图片最为匹配的特征向量，将这些最匹配的特征向量提取出来作为新的特征向量进行优化，最后得到的结果越来越像控制图片，这样就可以得到控制的梦境，具体可以看看下面的代码实现。

```
def objective_guide(dst, guide_features):
    x = dst.data[0].cpu().numpy().copy()
    y = guide_features.data[0].cpu().numpy()
    ch, w, h = x.shape
    x = x.reshape(ch,-1)
    y = y.reshape(ch,-1)
    A = x.T.dot(y) # compute the matrix of dot-products with guide features
    result = y[:,A.argmax(1)] # select ones that match best
    result = torch.Tensor(np.array([result.reshape(ch, w, h)], dtype=np.
    float)).cuda()
    return result
```

如果输入的控制图片是一张小猫的图片，最后通过 Deep Dream 能够得到图 7.16所示的图片，可以看到图片中有一些猫的头，猫的眼睛和猫的鼻子，是不是很神奇呢?

图 7.16 对猫做 Deep Dream

7.2.4 总结

通过上面的例子，我们不仅再次了解到了预训练网络的有用之处，同时学会了不仅仅只有网络的参数能够更新，输入的图片也能进行更新。除此之外，还了解到了如何将 backward 作用于标量上，如何手动更新参数，以及如何制作出 Deep Dream 效果之后的图片。

7.3 实例三——Neural-Style：使用 PyTorch 进行风格迁移

在上面两个实例中，我们都使用了预训练的卷积神经网络：一个是用来做迁移学习，节约计算资源，另外一个是用它来生成 Deep Dream 效果的图片，这一节我们将会使用预训练的卷积神经网络来实现艺术家的风格迁移，让我们自己的图片看着像是被艺术家画出来的一样。

7.3.1 背景介绍

前段时间有一个叫做 prisma 的 App 很火，就算各位读者没有用过，也一定看见别人使用过这个软件，下面是用这个软件得到的一个效果图，如图 7.17所示。

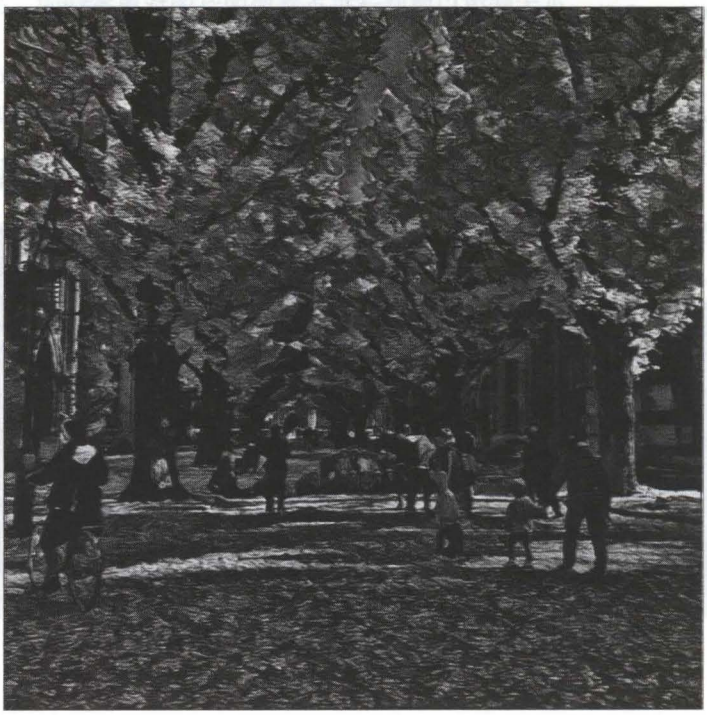


图 7.17 prisma 效果图

App 宣传的卖点是让你的照片拥有名家风格，比如毕加索、梵高等。其实这个软件背后的原理就是这里我们要讲的风格迁移，这个算法是 2015 年由 Leon A.Gatys, Alexander S.Ecker 等人在其论文中提出的，感兴趣的读者可以去阅读一下论文（https://arxiv.org/abs/1508.06576），这篇论文提出的算法就叫做 Neural-Style，里面主要涉及的

是卷积神经网络。

Neural-Style 也叫做 Neural-Transfer，是一个神奇的算法，输入一张图片，然后选择一种艺术风格图片，算法会将它们两者融合在一起，让输入的图片达到这种艺术的效果，如图 7.18所示。我们将一张海龟的图片输入，然后选择一种抽象派的风格，最后通过算法融合，得到了一张抽象的海龟图片。

图 7.18 融合艺术效果

7.3.2 原理分析

通过前面对于背景知识的介绍，可以知道需要实现的东西很清楚，就是要将两张图片融合在一起，但是这只是一种抽象的表达，我们需要准确地定义怎么样才算融合在一起。一张图主要由两个部分组成：一是内容；二是风格，所以论文中提出了两个度量方式：第一个就是在内容上相似程度的度量；第二个是风格上相似的度量。通过这样的定义，我们就明确地知道需要优化的目标是什么了。首先需要让融合图片和原始图片有尽可能高的相似度，或者说尽可能低的差异性；同时也需要让融合图片和风格图片在风格上尽可能相近。虽然我们知道了需要优化的目标是内容上的相似度和风格上的相似度，但是仍然没有具体的量化方式来处理，所以接下来就需要具体定义内容上相似和风格上相似的数学表达。

1. 内容差异

内容的差异性该如何定义呢？最简单的想法就是将两张图片每个像素点逐一进行比较，可以看作是均方误差，也就是求一下差，然后计算平方的和。但是通过第 6 章“生成对抗网络”了解到这种定义图片相似度误差的方式是很不好的，因为这样把握的其实是单个像素点的影响，而弱化了整体图片的影响，所以我们需要有更好的方式来定义内容误差。

前面一个实例中我们知道 Deep Dream 通过将图片输入预训练的网络来提取图片的特征，那么我们可不可以比较图片通过预训练网络提取的特征来代替直接比较图片呢？答案当然是可以的，特征可以看做是更高维度的图片，所以比较特征比直接比较图片有更好的效果。

我们让 C_{nn} 表示一个只含前面卷积提取特征部分的预训练网络，让 X 表示任何输入的图片，那么 $C_{nn}(X)$ 表示输入图片经过预训练网络每层提取出来的特征图的集合，每个特征图都是一个三维的矩阵，让 $F_{XL} \in C_{nn}(X)$ 表示第 L 层网络提取出来的特征图，大小是 $h \times w \times d$，我们可以将这个矩阵展开成一维的向量，那么这个向量的大小是 $h * w * d$，可以认为这张输入的图片 X 在网络第 L 层的内容就是 F_{XL}。

我们需要比较两张图片的内容差异，这两张图片大小应该是相同的，比如 Y 是另外一张图片，就可以定义这两张图片在 L 层的内容差异如下：

$$D_C^L(X,Y) = ||F_{XL} - F_{YL}||^2 = \sum_i (F_{XL}(i) - F_{YL}(i))^2 \tag{7.10}$$

其中 $F_{XL}(i)$ 表示网络第 L 层输出的特征图展开的向量的第 i 个元素。

我们该如何定义风格的差异性呢？这并没有内容差异性定义得那么直观，这也是这篇论文提出的创新点，通过引入 Gram 矩阵来表示图片的风格，据此再来计算风格的差异，首先我们介绍如何用 Gram 矩阵来定义图片风格，接着讲一下如何通过 Gram 矩阵来定义风格差异。

2.Gram 矩阵

首先各位读者都能想到要做的第一件事情就是将图片输入一个预训练的卷积神经网络提取出特征图，将图片空间投影到高维，这是前面使用过很多次的方法，因为直接在原始图片空间上做处理会损失掉很多的特征。前面我们用 $F_{XL}(i)$ 表示网络第 L 层输出的特征图展开的向量中的第 i 个元素，如果特征图不展开，我们知道这是一个大小为 $h \times w \times d$ 的矩阵，所以我们用 F_{XL}^k 表示这个大小为 $h \times w \times d$ 特征图中第 k 层厚度的矩阵，大小为 $h \times w$。

那么如何来定义 Gram 矩阵呢？首先 Gram 矩阵的大小是由特征图的厚度 d 决定的，等于 $d \times d$，每一个 Gram 矩阵中的元素，也就是 $Gram(i,j)$ 等于多少呢？先把特征图中厚度第 i 层和厚度第 j 层取出来，这样就得到了两个 $h \times w$ 的矩阵，分别表示成 F_{XL}^i 和 F_{XL}^j，然后将这两个矩阵对应元素相乘求和，这样就得到了 $Gram(i,j)$，可以用下面的公式来表示：

$$G_{XL}(i,j) =< F_{XL}^i, F_{XL}^j >= \sum_k F_{XL}^i(k) \cdot F_{XL}^j(k) \tag{7.11}$$

通过上面这个方式可以定义 Gram 矩阵中所有的元素，这样 Gram 矩阵中每个元素都可以理解为由特征图中的 i 层和 j 层矩阵相关性的表示，用其定义为图片在网络为 L 层输出的风格。

3. 风格差异

定义完了图片的风格之后，风格的差异就很简单了，就是两幅图的 Gram 矩阵的差异，就像内容的差异的计算方法一样，可以用下面的公式计算 L 层风格的差异。

$$D_S^L(X,Y) = ||G_{XL} - G_{YL}||^2 = \sum_{k,l}(F_{XL}(k,l) - F_{YL}(k,l))^2 \tag{7.12}$$

在风格转换中我们需要最小化几层内容差异 $D_C(X,C)$ 和几层风格差异 $D_S(X,S)$，所以我们的目标函数是这两者的和，对其求梯度可以用下面的公式表示：

$$\nabla(X,S,C) = \sum_{L_C} w_{CL_c} \cdot \nabla^{L_c}(X,C) + \sum_{L_s} w_{SL_s} \cdot \nabla^{L_s}(X,S) \tag{7.13}$$

其中 L_C 和 L_S 表示内容和风格分别需要几层的输出，这是一个可以根据所需要的效果任意设定的参数，w_{CL_C} 和 w_{SL_S} 表示内容上和风格上赋予的权重，也是可以根据需要的效果任意设置。

最后通过反向传播和梯度下降法能够优化图片中的像素点，实现风格转换的效果。

$$X \leftarrow X - \alpha\nabla(X,S,C) \tag{7.14}$$

7.3.3 代码实现

前面的部分有太多的数学公式，如果你仍然没有太理解风格转换的原理，没有关系，通过代码实现的部分会能够重新理解那些用数学符号表达的公式。

1. 内容差异

首先我们实现内容差异的代码，这个部分原理比较简单，代码也相对容易。

```
class Content_Loss(nn.Module):
    def __init__(self, target, weight):
        super(Content_Loss, self).__init__()
        self.weight = weight
        self.target = target.detach() * self.weight
        self.criterion = nn.MSELoss()

    def forward(self, input):
        self.loss = self.criterion(input * self.weight, self.target)
```

```
10          out = input.clone()
11          return out
12
13      def backward(self, retain_variabels=True):
14          self.loss.backward(retain_variables=retain_variabels)
15          return self.loss
```

这里注意，定义的 loss 不像我们之前定义的 loss 一样，因为之前的 loss 在 PyTorch 中都已经实现了，但是这里我们需要自定义新的 loss，所以如果想要像之前一样定义，需要重写 backward。这里使用网络模型的定义方式，把这个部分当成一个网络层，下面我们来完整地解释整个定义方式。

图片经过预训练的卷积层得到 L 层的特征图 F_{XL}，最后输出与基准内容图片的加权内容距离 $w_{CL} \cdot D_C^L(X,C)$，所以初始化中需要传入基准图片和权重，上面的代码中传入的基准图片为 target，权重为 weight。

接着在初始化中，通过 `target.detach()` 将基准图片从计算图中分离出来，看做一个常量。定义 `nn.MSELoss()` 为了计算 L 层输出的两个特征图的距离 $\|F_{XL}-F_{YL}\|^2$。

将上面定义的内容差异看做是一个网络层结构，在 forward 的实现中，将输入的特征图和基准特征图求出内容距离保存为 `self.loss`，然后用 clone 将输入复制成输出，传出网络。可以看到特征图传到这一层网络结构之后再传出，输入和输出是完全相同的，只是经过这层网络之后，会计算出一个内容的差异被保存起来。

最后定义 backward，定义这一层网络是如何进行反向传播的。该层网络相当于把输入直接转换成输出，然后进入下一层网络，如果不重新定义反向传播，那么这一层就相当于一个恒等变换，是没有办法将计算出来的内容差异反向传播回去更新参数的。

在重新定义的 backward 中，我们将内容距离，也就是 `self.loss` 进行反向传播，这里注意一个小细节，`retain_variables=True`，正如前面讲过的，这个变量是为了在反向传播之后保留计算图，因为网络中会有多层的内容差异和风格差异，所以还会做多次反向传播。

2. 风格差异

首先需要定义好 Gram 矩阵，根据 7.11 的公式我们可以将特征图 F_{XL} 重新排列成 $(d, m \times n)$，用 K 来表示，那么上面这个公式可以改写成下面这样：

$$Gram(i,j) = \sum_s K(i,s) \cdot K(j,s) \tag{7.15}$$

$$= \sum_s K(i,s) \cdot K^T(s,j) \tag{7.16}$$

$$= K \cdot K^T(i,j) \tag{7.17}$$

所以可以得到 $Gram = K \cdot K^T$，得到了这个结论，就能很容易地写出 Gram 矩阵的定义。

```
class Gram(nn.Module):
    def __init__(self):
        super(Gram, self).__init__()

    def forward(self, input):
        a, b, c, d = input.size()
        feature = input.view(a * b, c * d)
        gram = torch.mm(feature, feature.t())
        gram /= (a * b * c * d)
        return gram
```

在 forward 中，我们将输入的特征用 view 重新排列，然后用 `torch.mm` 做矩阵乘法，最后标准化得到 Gram 矩阵。

定义好了 Gram 矩阵，我们就可以定义风格差异了，方法和内容差异几乎一模一样，代码如下所示。

```
class Style_Loss(nn.Module):
    def __init__(self, target, weight):
        super(Style_Loss, self).__init__()
        self.weight = weight
        self.target = target.detach() * self.weight
        self.gram = Gram()
        self.criterion = nn.MSELoss()

    def forward(self, input):
        G = self.gram(input) * self.weight
        self.loss = self.criterion(G, self.target)
        out = input.clone()
        return out

    def backward(self, retain_variabels=True):
        self.loss.backward(retain_variables=retain_variabels)
        return self.loss
```

3. 建立模型

有了前面内容差异和风格差异的定义，就可以据此来建立模型了。

首先，会使用预训练的 19 层 VGG 网络，去掉最后的全连接层，只保留前面的卷积层。

```
vgg = models.vgg19(pretrained=True).features
```

接着，需要确定哪几层的网络输出作为内容和风格的输出层，通过下面的代码来定义。

```
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']
```

这里我们只使用第四层卷积层作为内容层的输出，使用 1~5 层的卷积层输出作为风格层的输出，需要重构我们的网络模型，将内容差异和风格差异加入到原先的网络层当中，可以通过下面的代码来达到所需要的目的。

```
for layer in cnn:
    if isinstance(layer, nn.Conv2d):
        name = 'conv_' + str(i)
        model.add_module(name, layer)

        if name in content_layers_default:
            target = model(content_img)
            content_loss = loss.Content_Loss(target, content_weight)
            model.add_module('content_loss_' + str(i), content_loss)
            content_loss_list.append(content_loss)

        if name in style_layers_default:
            target = model(style_img)
            target = gram(target)
            style_loss = loss.Style_Loss(target, style_weight)
            model.add_module('style_loss_' + str(i), style_loss)
            style_loss_list.append(style_loss)

        i += 1
    if isinstance(layer, nn.MaxPool2d):
        name = 'pool_' + str(i)
```

```
            model.add_module(name, layer)

        if isinstance(layer, nn.ReLU):
            name = 'relu' + str(i)
            model.add_module(name, layer)
```

我们遍历网络中的每一层，然后判断一下，如果这一层使我们要求的内容输出层或者是风格输出层，我们就可以将前面定义的 ContentLoss 和 Style_loss 添加到其后面，这样遍历完整个网络结构之后，便构建出了我们需要的网络结构。

4. 运行代码

建立完模型就可以反向传播更新参数了，需要注意的是更新的参数并不是网络中的参数，而是输入图片的像素，所以首先需要申明一下参数。

```
def get_input_param_optimier(input_img):
    """
    input_img is a Variable
    """
    input_param = nn.Parameter(input_img.data)
    optimizer = optim.LBFGS([input_param])
    return input_param, optimizer
```

这里使用 nn.Parameter 将输入图片变成计算图中的叶子节点，从而能够进行梯度下降更新参数，然后论文推荐使用二阶收敛方法 L-BFGS，不太了解的读者可以自行查阅相关资料，这里就不再展开说。

接着就可以进行模型的训练了。

```
def run_style_transfer(content_img, style_img, input_img, num_epoches=300):
    print('Building the style transfer model..')
    model, style_loss_list, content_loss_list = get_style_model_and_loss(
        style_img, content_img)
    input_param, optimizer = get_input_param_optimier(input_img)

    print('Opimizing...')
    epoch = [0]
    while epoch[0] < num_epoches:

        def closure():
```

```
            input_param.data.clamp_(0, 1)

            model(input_param)
            style_score = 0
            content_score = 0

            optimizer.zero_grad()
            for sl in style_loss_list:
                style_score += sl.backward()
            for cl in content_loss_list:
                content_score += cl.backward()

            epoch[0] += 1
            if epoch[0] \% 50 == 0:
                print('run {}'.format(epoch))
                print('Style Loss: {:.4f} Content Loss: {:.4f}'.format(
                    style_score.data[0], content_score.data[0]))
                print()

            return style_score + content_score

        optimizer.step(closure)

        input_param.data.clamp_(0, 1)

    return input_param.data
```

网络通过对所有的内容差异和风格差异进行多次反向传播来更新参数，注意一下里面定义了 closure，这是因为我们使用 L-BFGS 算法，所以更新参数的时候需要在 step 里面调用。最后还需要注意的一点就是每次我们都会将更新之后的参数，也就是最后要输出的风格融合的图片中的数据的大小进行重新裁剪，因为更新的过程中可能会超过图片数据允许的大小，一般是 0～255。

5. 结果

我们输入内容图片和风格图片，经过我们定义的网络进行参数更新之后得到如图 7.19所示的图片。

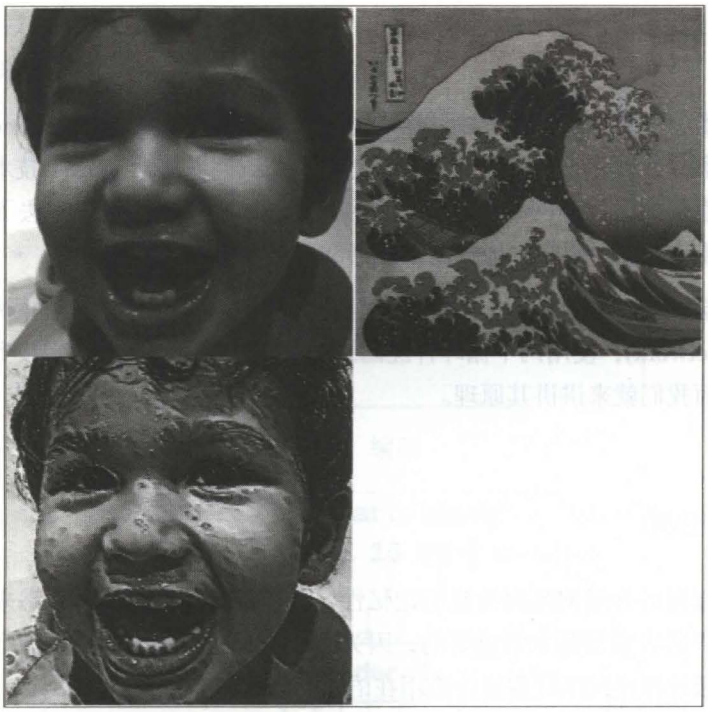


图 7.19 图片风格融合结果

7.3.4 总结

通过这一节，我们利用 PyTorch 实现了基本的风格转移算法，通过预训练的卷积网络提取出更高维度的图片的内容和风格，最后通过定义内容损失函数和风格损失函数进行反向传播更新参数，实现了图片风格融合的算法。从该算法中，我们也了解到了很多独特的定义方式，以及预训练网络的神奇之处。

7.4 实例四——Seq2seq：通过 RNN 实现简单的 Neural Machine Translation

前面三个例子都是讲卷积神经网络在不同方面的应用，最后一个例子我们讲一下循环神经网络如何应用在自然语言处理中的机器翻译上，介绍如何教会神经网络进行法语和英语的翻译。

7.4.1 背景介绍

随着深度学习的崛起，基于神经网络的机器翻译技术（Neural Machine Translation，NMT）也取得了长足的进步，迅速替代了之前的主流翻译技术。NMT 技术使用了端到端的方式，直接通过神经网络建立源语言和目标语言的对应关系，省去了传统方法烦琐的预处理流程，同时也极大地提高了机器翻译的准确率。

而引发这一切成功的是一个简单却又强大的想法，即序列到序列的网络（sequence to sequence network），使用两个循环神经网络，将一个语言序列直接转换到另外一个语言序列，下面我们就来讲讲其原理。

7.4.2 原理分析

前面讲过循环神经网络因为具有记忆性，所以能够根据前面的状态来预测后面的状态，这对于语言模型是非常重要的，因为同一个词在不同的语境下面有着不一样的意思，所以循环神经网络特别适合应用在语言模型中。

序列到序列的模型是循环神经网络的升级版，其联合了两个循环神经网络：一个神经网络负责接收要源句子；另外一个循环神经网络负责将句子输出成翻译的语言。这两个过程分别被称为编码（encoder）和解码（decoder）的过程。如果还记得 6.1.1“自动编码器”那一节的内容，你就会记得那里也有一个编码和解码的过程，它们的原理是相似的，下面我们来讲讲这两个过程。

1. 编码（encoder）

编码过程实际上使用了循环神经网络记忆的功能，通过上下文的序列关系，将词向量依次输入网络。对于循环神经网络，我们知道每一次网络都会输出一个结果，但是编码（encoder）不同之处在于，其只保留最后一个隐藏状态，相当于将整句话浓缩在一起，将其存为一个内容向量（context）供后面的解码器（decoder）使用。编码可以由如图7.20所示的示意图更加直观地表示。

2. 解码（deocder）

解码和编码网络结构几乎是一样的，唯一不同的就是在解码过程中，是根据前面的结果来得到后面的结果，什么意思呢？因为在编码过程输入一句话，这一句话就是一个序列，而且序列中的每个词都是已知的，而解码过程相当于什么都不知道，首先需要一个标识符表示一句话的开始，然后将其输入网络得到第一个输出作为这句话的第一个词，接着通过得到的第一个词作为网络的下一个输入，得到的输出作为第二个词，

不断循环，通过这种方式来得到最后网络输出的一句话。如果感觉有些疑惑，可以通过图 7.21所示的示意图来理解。

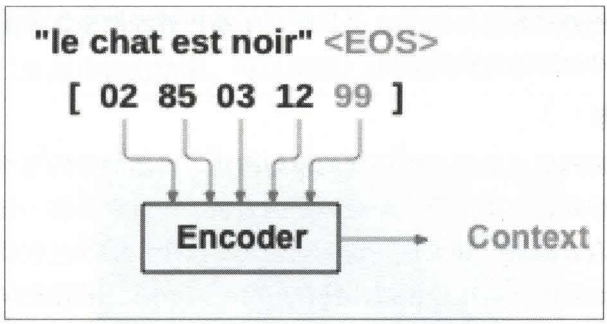


图 7.20 编码

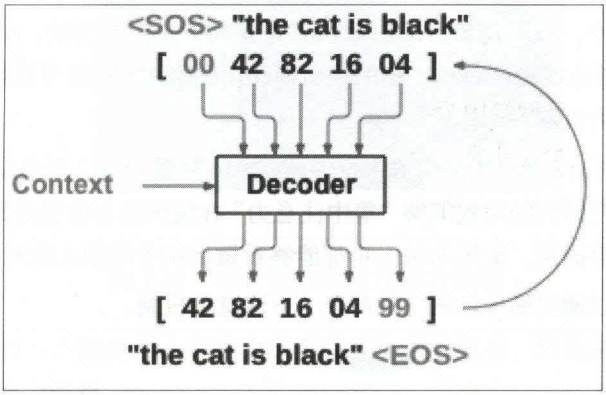


图 7.21 解码

从 6.1.1 节我们知道，卷积神经网络也可以作为一个编码器和解码器，这里使用循环神经网络作为编码器和解码器的好处是什么呢？这是因为语句存在序列关系，对于同一个词，其前面的词语不同，那么这个词所代表的含义也将不同，而循环神经网络有记忆的这一特性能够很好地处理这种关系。

那么使用序列到序列（seq2seq）这种网络结构的原因又是什么呢？第一，翻译的每句话的输入长度和输出长度一般来讲都是不同的，而序列到序列的网络结构的优势在于不同长度的输入序列能够得到任意长度的输出序列；第二，在翻译中，虽然有一些词能够一一对应，比如中文中的“猫”对应英文中的“cat”，但是对于一句话，每个词并不一定是一一对应的，比如一句中文“你今天吃了什么?”，翻译成英文就是“what do you eat today?”，这里整个语句的对应关系就是颠倒的，所以使用序列到序列的模型，首先将一句话所有的内容压缩成一个内容向量，然后通过一个循环网络不断地内容提取出来，形成一句新的话。

同时序列到序列的模型不仅仅能够用在机器翻译上，这个想法还能够用在图片字幕上，将图片通过卷积网络提取特征，然后使用序列模型得到图片中的内容描述，本质上跟上面的编码解码过程是一致的，只是使用了卷积神经网络作为编码器。这样的模型本质上都是编码和解码的网络结构。

3. 注意力机制

通过编码和解码形成的序列到序列的模型构建了机器翻译的基本框架，但是在实际中表现往往不够好，这里面很大的原因是因为在编码过程会将一句话的内容压缩成一个固定大小的内容向量，如果这一句话比较长，这个压缩的过程就容易失去这句话中的信息，同时如果两句话比较相似，只有一两个词不同，压缩成内容向量过于相似，会导致在解码的过程表现不够好。

因为上面问题的存在，我们希望能够找到一个机制来解决这个问题。在之前的序列到序列的模型中，我们将最后的隐藏状态保存成一个内容向量，而每一步的输出我们并没有使用，那么能不能将每一步的输出都利用起来呢？答案当然是可以的，这就是下面要讲的注意力机制。

注意力机制（attention）是由 Bahdanau 在 2014 年提出的，听名字就知道，注意力机制就是让网络在解码的时候能够“集中注意力”在编码输出的某些部分上，而不仅仅依赖于简单的内容向量，解码的每一步都能够看做是在考虑输入序列的不同部分。

下面我们可以通过简单的图 7.22解释一下注意力机制。

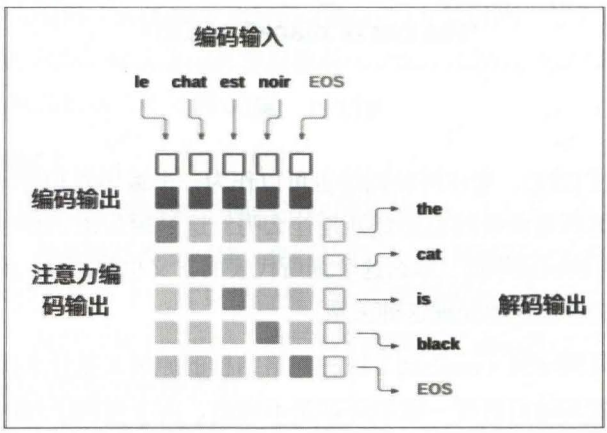


图 7.22　注意力机制

图 7.22 就是注意力机制的简单图示，对于每个解码的输出，都是由编码的输出和权重一起决定的，图上蓝色表示编码的输出，橙色表示权重，颜色越深表示权重越大。可以看出第一行中最深的颜色出现在最左边，说明第一个解码的输出更多地依赖于编码输出的前面部分。

所以说注意力机制就是要赋予权重，那么权重是如何得到的呢？权重并不是预设的，而是通过网络学习去更新的。解码过程的每一步都会有一个权重，权重的计算依赖于解码过程的输入和隐藏状态来计算的，并不依赖于编码过程的输出，而编码过程的输出需要和权重结合起来输出结果，可以用图7.23来简单表示。

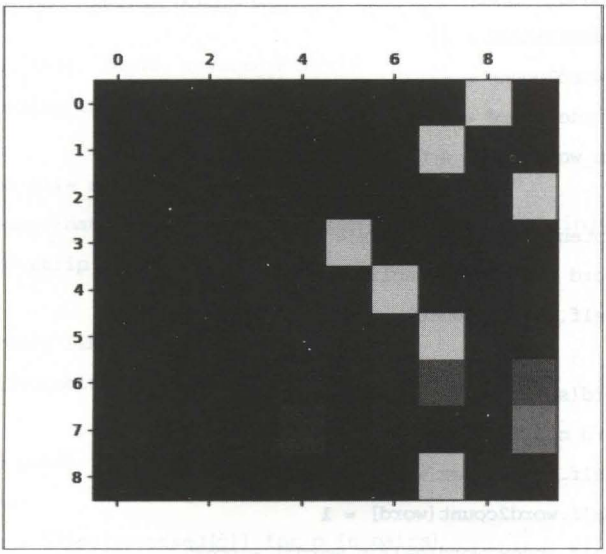


图 7.23 与权重结合的注意力机制

解码过程中每一步都将输入和隐藏状态生成注意力权重，由此和编码过程的输出结合起来形成注意力机制。

4.teacher forcing

最后提一点训练中的技巧能够加快收敛速度，那就是 teacher forcing。这个方法也特别简单，就是在解码的过程，不再是把前一步得到的输出当做下一步的输入，而是将正确的词作为输入，因为前一步的输出并不一定是正确的，所以使用 teacher forcing 能够保证作为输入的永远是正确的词，这样就能够加快收敛的速度。

7.4.3 代码实现

前面的原理分析已经完全地讲解了这一节内容背后的原理，下面会通过代码来加深理解。

1. 数据预处理

首先通过这个网址下载数据集：https://download.pytorch.org/tutorial /data.zip，然后将其解压放到主目录下面。每个单词没有办法直接作为网络的输入，所以需要给每个

不同的单词一个下标作为它的标签，可以通过下面的代码实现。

```
class Lang(object):
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

上面的部分首先建立一个字典类，“SOS”和“EOS”表示一句话的开始标志和结束标志，这两个字符先给它们 0 和 1 的标签，然后通过下面两个函数 addSentence 和 addWord 去分析每句话，如果一句话中出现了没有标记过的词，就在字典中建立一个新的对应关系。

为了简化，我们通过上面的代码将 Unicode 的字符都转化成 ASCII，让所有的字母都是小写，去掉所有的标点。

```
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn')

def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
```

```
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

因为下载下来的数据集是一对一对地放在 txt 文档中的，所以需要定义一个函数来读取 txt 文档中的内容并保存，我们可以定义下面的函数来完成这件事：

```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/\%s-\%s.txt' \% (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

为了简化计算，我们不将整个数据集作为训练集，因为这样需要的训练时间太长，只保留一部分以某些前缀开头的句子和长度小于 10 的句子作为训练集进行训练，所以可以通过下面的方式过滤掉不满足条件的数据集。

```
eng_prefixes = ("i am ", "i m ", "he is", "he s ", "she is", "she s",
                "you are", "you re ", "we are", "we re ", "they are",
                "they re ")

def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)
```

```
9
10 def filterPairs(pairs):
11     return [pair for pair in pairs if filterPair(pair)]
```

接着将前面定义的函数组合在一起构成一个新的函数，这个函数首先从文本中读取数据，然后过滤掉不满足条件的数据，接着将要进行翻译的两种语言一次放在不同的字典下，然后将所有出现过的词与一个标签对应起来，保存成两个字典。

```
1  def prepareData(lang1, lang2, reverse=False):
2      input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
3      print("Read \%s sentence pairs" \% len(pairs))
4      pairs = filterPairs(pairs)
5      print("Trimmed to \%s sentence pairs" \% len(pairs))
6      print("Counting words...")
7      for pair in pairs:
8          input_lang.addSentence(pair[0])
9          output_lang.addSentence(pair[1])
10     print("Counted words:")
11     print(input_lang.name, input_lang.n_words)
12     print(output_lang.name, output_lang.n_words)
13     print(random.choice(pairs))
14     return input_lang, output_lang, pairs
```

通过三个函数依次对输入的一句话做预处理：第一个函数将一句话里面出现的单词都转化成对应的标签；第二个函数在第一个函数的基础上，将一句话转换成对应的标签序列；第三个函数依据于前面两个定义好的函数，将一组对应的翻译语言分别转化成两个标签序列。

```
1  def indexesFromSentence(lang, sentence):
2      return [lang.word2index[word] for word in sentence.split(' ')]
3
4  def tensorFromSentence(lang, sentence):
5      indexes = indexesFromSentence(lang, sentence)
6      indexes.append(EOS_token)
7      result = torch.LongTensor(indexes)
8      return result
9
10 def tensorFromPair(input_lang, output_lang, pair):
```

```
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return input_tensor, target_tensor
```

最后结合前面所有的函数，自定义一个数据集继承于 Datatset 基类，定义好__getitem__和__len__两个函数。

```
lass TextDataset(Dataset):
    def __init__(self, dataload=prepareData, lang=['eng', 'fra']):
        self.input_lang, self.output_lang, self.pairs = dataload(
            lang[0], lang[1], reverse=True)
        self.input_lang_words = self.input_lang.n_words
        self.output_lang_words = self.output_lang.n_words

    def __getitem__(self, index):
        return tensorFromPair(self.input_lang, self.output_lang,
                              self.pairs[index])

    def __len__(self):
        return len(self.pairs)
```

通过运行上面的程序，可以得到图 7.24所示的结果。

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10853 sentence pairs
Counting words...
Counted words:
fra 4489
eng 2925
['il est desormais dans une situation tres difficile .', 'he is now in a very difficult situation .']
```

图 7.24 运行结果

可以发现通过过滤将数据大小从 135842 减少到了 10853，得到英语和法语的单词数目分别是 2925 和 4489，最后列出了一组英语和法语翻译的文字。

2. 模型

下面建立序列到序列的模型。先从最基本的编码解码模型开始，建立两个简单的循环神经网络。

首先建立一个编码器（encoder）。

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
```

```
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        input = input.unsqueeze(1)
        embedded = self.embedding(input)  # batch, hidden
        output = embedded.permute(1, 0, 2)
        for i in range(self.n_layers):
            output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

初始化中定义了隐藏状态的大小及网络层数，但是这里的网络层数与之前我们讲的网络层数不太相同，这里的层数不是定义在循环神经网络中，而是定义在网络之外的。换句话说，如果定义在循环神经网络中，那么有多少层就有多少个隐藏状态，而定义在循环网络外面，那么只有一个隐藏状态，这个隐藏状态会继续往后面的网络层中传递。在前向传播中，将输入的标签序列转化为词向量序列，接着进入网络输出结果和隐藏状态，这里的循环表示定义在网络外的层数。

接着定义好解码器（decoder）。

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, n_layers=1):
        super(DecoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
```

```
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax()

    def forward(self, input, hidden):
        output = self.embedding(input)  # batch, 1, hidden
        output = output.permute(1, 0, 2)  # 1, batch, hidden
        for i in range(self.n_layers):
            output = F.relu(output)
            output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

解码的过程跟编码的过程几乎是一样的，唯一不同的是会多定义一个线性层作为输出层，将维度转换到单词的数目，因为最后是根据所有输出中的最高概率来确定输出的。

编码和解码的整个过程可以表示成图 7.25。

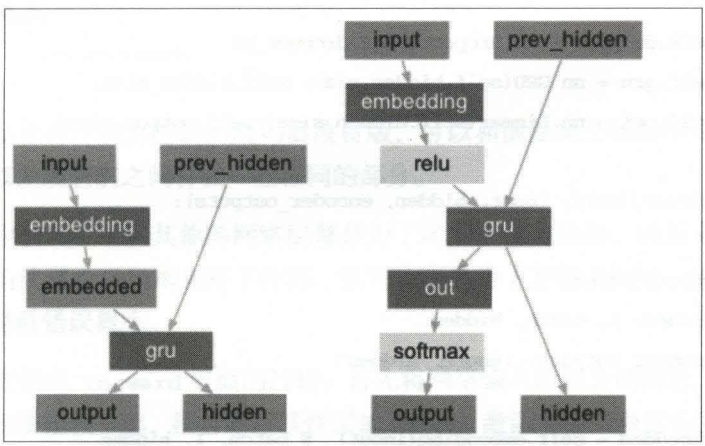


图 7.25 编码和解码

左边是编码过程，右边是解码过程，都是由简单的循环神经网络所构成的。

建立完最简单的模型之后，我们可以将注意力机制添加到解码器中建立更复杂的模型。

因为注意力机制是添加到解码器中的，所以编码器不需要做任何修改，只需要重新建立一个含有注意力机制的解码器。

```
class AttnDecoderRNN(nn.Module):
    def __init__(self,
                 hidden_size,
                 output_size,
                 n_layers=1,
                 dropout_p=0.1,
                 max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size
        )
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        '''
        input: batch, 1
        hidden: 1, batch, hidden
        encoder_outputs: length, hidden
        '''
        embedded = self.embedding(input)  # batch, 1, hidden
        embedded = self.dropout(embedded)
        embedded = embedded.squeeze(1)  # batch, hidden

```

```
        attn_weights = F.softmax(
            self.attn(torch.cat((embedded, hidden[0]), 1)))
        # batch, max_length
        encoder_outputs = encoder_outputs.unsqueeze(0)
        # batch, max_length, hidden
        attn_applied = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
        # batch, 1, hidden
        output = torch.cat((embedded, attn_applied.squeeze(1)), 1)
        # batch, 2xhidden
        output = self.attn_combine(output).unsqueeze(0)
        #1, batch, hidden

        for i in range(self.n_layers):
            output = F.relu(output)
            output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output.squeeze(0)))
        return output, hidden, attn_weights

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

上面就是含有注意力机制的解码器改良版，可以和前面定义的最基本的解码器对比一下，可以发现它们之间存在一些相同的部分。

初始化中引入了一些其他的网络层都是为了定义注意力机制，而最重要的注意力机制在上面的代码中已经添加好了注释，这些注释是为了方便了解每一步之后的数据大小的，可降低错误概率。

我们直接考虑 forward 里面的过程，首先将网络输入转化为词向量，然后将词向量和隐藏状态拼接在一起，接着通过线性层加 softmax 激活层输出固定长度的序列。这个序列就是注意力序列，每个数的大小表示注意力的重要程度，然后使用 torch.bmm 将编码过程的输出和注意力权重通过批矩阵乘法得到输出结果，最后将这个结果和网络输入拼接在一起，通过一个线性层将维度转化为循环神经网络接受的维度，将其作

为网络的输入，最后通过网络得到最后的输出。

通过上面的方式，我们就在解码的过程中添加了注意力机制。

3. 模型训练

建立好网络之后就可以开始模型训练了，得益于 PyTorch 的动态图和命令式编程，所以在模型的训练中可以添加 for 循环，非常方便。

首先是编码的过程，先定义一个零序列，每一步编码之后的输出都填入这个零序列当中，最后将隐藏状态保存为解码过程的初始隐藏状态。

```
encoder_outputs = Variable(
    torch.zeros(MAX_LENGTH, encoder.hidden_size))
if torch.cuda.is_available():
    encoder_outputs = encoder_outputs.cuda()
encoder_hidden = encoder.initHidden()
for ei in range(in_lang.size(1)):
    encoder_output, encoder_hidden = encoder(
        in_lang[:, ei], encoder_hidden)
    encoder_outputs[ei] = encoder_output[0][0]

decoder_hidden = encoder_hidden
```

接着定义解码过程，这里分为两种情况：一种是不使用注意力机制，一种是使用注意力机制。

对于不使用注意力机制的训练过程，非常简单，每次将输入和隐藏状态传入解码器中，将得到的结果再传入解码器的下一步就可以了，在循环中判断一下，如果输入的是结束符，那么这句话就结束了，跳出这个循环。

```
for di in range(out_lang.size(1)):
    decoder_output, decoder_hidden = decoder(
        decoder_input, decoder_hidden)
    loss += criterion(decoder_output, out_lang[:, di])
    topv, topi = decoder_output.data.topk(1)
    ni = topi[0][0]

    decoder_input = Variable(torch.LongTensor([[ni]]))
    if torch.cuda.is_available():
        decoder_input = decoder_input.cuda()
    if ni == EOS_token:
```

```
        break
```

如果使用注意力机制，只需在解码的过程中传入编码的输出，其余部分是一样的。

```
for di in range(out_lang.size(1)):
    decoder_output, decoder_hidden, decoder_attention = attn_decoder(
        decoder_input, decoder_hidden, encoder_outputs)
    loss += criterion(decoder_output, out_lang[:, di])
    topv, topi = decoder_output.data.topk(1)
    ni = topi[0][0]

    decoder_input = Variable(torch.LongTensor([[ni]]))
    if torch.cuda.is_available():
        decoder_input = decoder_input.cuda()
    if ni == EOS_token:
        break
```

网络使用注意力机制，并训练了 20 次，能够得到图 7.26所示的误差效果图。

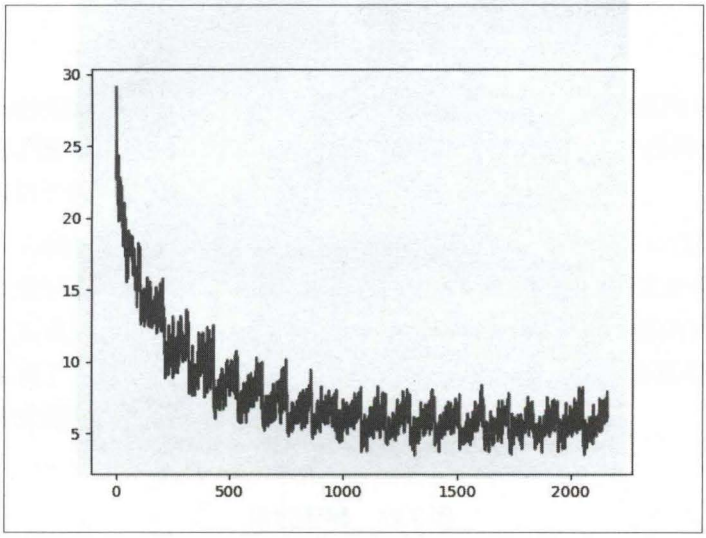


图 7.26 误差效果图

如果希望能够得到更好的效果，可以使用 teacher forcing，以及更加复杂的网络结构和注意力机制的多种变式。

4. 模型评估

训练得到的模型需要评估其效果如何，而评估的过程跟网络的训练过程是类似的，

不断将前一步网络得到的结果作为网络下一步的输入，直到最后输出一句话的终止标志或者是长度超过最大长度限制。

通过随机从文本中搜寻几句话，能够得到图 7.27所示的翻译效果，其中第一句话是待翻译的法文，第二句话是真实的英文翻译，第三句话是机器翻译得到的结果，可以发现通过简单序列到序列的模型，机器翻译能够取得一定的效果。

```
> tu es une etudiante .
= you are a student .
< you are a student . <EOS>

> je suis tout a coup fatiguee .
= i m suddenly tired .
< i m suddenly . . <EOS>

> elles sont mes amies .
= they are my friends .
< they are my friends . <EOS>

> tu es aveuglee par l amour .
= you are blinded by love .
< you are blinded by love . <EOS>

> je suis en train d utiliser cette tasse .
= i m using that cup .
< i m using cup cup cup . <EOS>

> tu es tres solitaire .
= you re very lonely .
< you re very lonely . <EOS>

> je ne vais pas m impliquer .
= i am not getting involved .
< i m not going to get . <EOS>

> vous etes tres contrariee .
= you re very upset .
< you re very upset . <EOS>

> vous m attirez beaucoup .
= i m very drawn to you .
< i m very drawn to you . <EOS>

> vous avez tellement tort .
= you re so wrong .
< you re wrong wrong . <EOS>
```

图 7.27　翻译效果

同时还可以将注意力的权重可视化，查看一下网络在翻译每个单词的时候到底注意力更加集中在哪些部分，如图 7.28所示。

可以看到在翻译一句话的时候，注意力也是不断集中于编码输出的不同区域。同时注意力机制还有一个很好的性质，即具有很强的解释性，能够帮助我们理解网络到底从哪里学习、学到了什么。

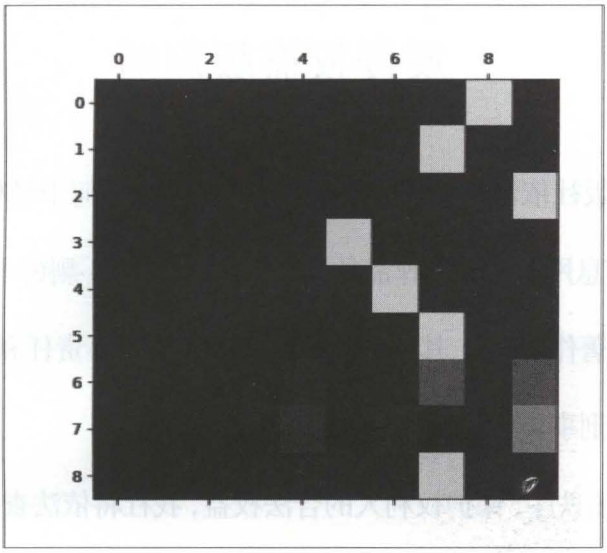


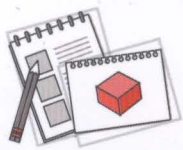
图 7.28 注意力的权重可视化

7.4.4 总结

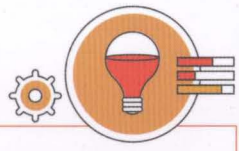
通过上面的部分，我们学会了如何使用序列到序列的模型去构建简单的机器翻译系统，同时也了解了注意力机制的原理，以及如何使用注意力机制来改善模型效果，感兴趣的读者可以尝试其他的语言翻译，比如中文到英文。

序列到序列模型的应用不仅限于此，还能够用于对话系统、问答系统等，如果读者感兴趣，可以尝试更多的序列到序列的模型应用。同时注意力机制近来也变得非常流行，谷歌最近发表了一篇论文，只使用注意力机制而不是用任何具体的网络结构来做机器翻译，达到了目前业界的最好水平，可以看出注意力机制有很多值得探寻的地方，期待读者们带着问题去阅读相关论文，找到更多有意义的应用。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版



深度学习入门之PyTorch



本书将以PyTorch为工具，从基础的线性回归开始，讲到时下前沿的生成对抗网络，并在其中穿插PyTorch的教学。学习完本书之后，读者能够大致了解深度学习的基本知识，基本掌握PyTorch的使用方法，知道如何根据实际问题搭建对应的深层网络结构，并进行调参得到较好的结果。

通过学习本书，可学到：

- ☑ 怎样才能快速入门深度学习领域。
- ☑ 怎么快速入门PyTorch，了解PyTorch的基本操作。
- ☑ 掌握多层神经网络、卷积神经网络、循环神经网络、自动编码器和生成对抗网络。
- ☑ 学习图像分类、迁移学习、自然语言处理等深度学习应用。
- ☑ 通过4个实战练习更进一步掌握深度学习在现实中的应用。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：孙学瑛
封面设计：李　玲

上架建议：人工智能>深度学习

ISBN 978-7-121-32620-2
9 787121 326202 >
定价：79.00元